VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Microsoft Robotics Studio – – programování robotů

Microsoft Robotics Studio – – Robots Programming

2010 David Turoň

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. David Turoň**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Microsoft Robotics Studio - programování robotů
Microsoft Robotics Studio - Robots Programming

Zásady pro vypracování:

Cílem práce je návrh a implementace úloh pro robota LEGO MINDSTORM NXT v prostředí Microsoft Robotics Developer Studio.
1. Seznamte se s MRDS a robotem LEGO MINDSTORM NXT.
2. Proveďte průzkum jiných existujících prostředí pro programování těchto robotů.
3. Navrhněte a naimplementujte vhodné úlohy pro tohoto robota.

Seznam doporučené odborné literatury:

Podle pokynů vedoucího diplomové práce.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **RNDr. Eliška Ochodková, Ph.D.**

Datum zadání: 20.11.2009
Datum odevzdání: 07.05.2010

doc. Dr.Ing. Eduard Sojka
vedoucí katedry

prof. Ing. Ivo Vondrák, CSc.
děkan fakulty

Souhlasím se zveřejněním této diplomové práce dle požadavků čl. 26, odst. 9 *Studijního a zkušebního řádu pro studium v magisterských programech VŠB-TU Ostrava.*

V Ostravě 30. dubna 2010 .....

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 30. dubna 2010 .....

Rád bych na tomto místě poděkoval všem, kteří mi s prací jakýmkoliv způsobem pomohli, zvláště pak RNDr. Elišce Ochodkové, Ph.D. za cenné rady.

Abstrakt

Tato práce pojednává o možnostech programování robotů LEGO Mindstorms NXT v nástroji Microsoft Robotics Developer Studio (MRDS). Krom tohoto nástroje jsou popsány nástroje RobotC a LEGO Mindstorms (NXT–G). Jsou zde vyčteny výhody a nedostatky jednotlivých nástrojů a vytvořena aplikace s řešením bludiště pro robota LEGO NXT.

Klíčová slova: MRDS, LEGO NXT, VPL, RobotC, robot

Abstract

This thesis describes the possibilities of programming LEGO Mindstorms NXT robots with Microsoft Robotics Developer Studio (MRDS). Besides this tool, two other tools are described here, namely RobotC and LEGO Mindstorms (NXT–G). There are lists of pros and cons for particular tools and a ready–made application solving the labyrinth for LEGO NXT robot.

Keywords: MRDS, LEGO NXT, VPL, RobotC, robot

Seznam použitých zkratek a symbolů

API	–	Application Programming Interface
CLR	–	Common Language Runtime
CRR	–	Concurrency and Coordination Runtime
DSS	–	Decentralized Software Services
DSSME	–	DSS Manifest Editor
EOPD	–	Electro – Optical Proximity Detector
HTTP	–	Hypertext Transfer Protocol
I^2C	–	Inter – Integrated Circuit
IDE	–	Integrated Development Environment
IR	–	Infra Red
LED	–	Light – Emitting Diode
MRDS	–	Microsoft Robotics Developer Studio
PDF	–	Portable Document Format
REST	–	Representational State Transfer
RF	–	Radio frequency
RIS	–	Robotics Invention System
RPC	–	Remote Procedure Call
SOAP	–	Simple Object Access Protocol
SVN	–	Subversion
URI	–	Uniform Resource Identifier
URL	–	Uniform Resource Locator
USB	–	Universal Serial Bus
VPL	–	Visual Programming Language
VSE	–	Visual Simulation Environment
WWW	–	World Wide Web
XML	–	Extensible Markup Language

Obsah

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

Nástroje na programování robotů lze obecně dělit do dvou skupin, první skupina nahrává programy do paměti robota a druhá skupina využívá vzdálené volání procedur (RPC) robota. Představíme zástupce z obou skupin. Z první LEGO Mindstorms NXT a RobotC, z druhé Microsoft Robotics Developer Studio (MRDS). Dále lze dělit nástroje na nástroje s grafickým programovacím jazykem a na nástroje se strukturovaným programovacím jazykem. Do nástrojů s grafickým jazykem patří LEGO Mindstorms NXT, do nástrojů se strukturovaným jazykem RobotC. MRDS patří do obou skupin.

Cílem této práce je navrhnout a implementovat úlohy pro robota LEGO Mindstorms NXT v prostředí MRDS. Tato práce popisuje architekturu MRDS, z čeho se skládá a jak pomocí MRDS programovat roboty. Ačkoli MRDS umí programovat nespočet druhů robotů, zaměříme se pouze na roboty LEGO NXT. Ukážeme zde některé možnosti programování LEGO NXT v prostředích RobotC a LEGO Mindstorms NXT, jejich klady a zápory, možnosti kooperace mezi jazyky. Ve Visual Programming Language (VPL) vytvoříme kopie jednoduchých úloh napsaných v RobotC pro porovnání mezi grafickým a strukturovaným programovacím jazykem. Popíšeme taky senzory, které LEGO NXT využívá. Nakonec ukážeme tvorbu aplikace pro prohledávání bludiště a nalezení nejkratší cesty s robotem LEGO NXT.

2 LEGO Mindstorms

Každý z nás jistě prošel obdobím, kdy dostal od rodičů do ruky nějakou stavebnici Merkur nebo LEGO. Mohl svou tvůrčí činností vytvořit nějakou vlastní hračku. Buď šlo o hračku z nějaké předlohy nebo o vlastní nápad. Tato hračka mohla umět spoustu věcí, ale pokaždé jí něco chybělo. To něco byla inteligence. Společnost LEGO šla ovšem s dobou. Chybějící inteligenci doplnila pomocí programovatelných procesorů skrývajících se v útrobách těchto hraček. LEGO Mindstorms je řada programovatelných robotických stavebnic, které firma LEGO vyrábí.

Nejprve v roce 1998 vytvořila první programovatelnou hračku pojmenovanou Robotics Invention System (RIS), později v roce 2006 vytvořila verzi LEGO NXT, která byla použita v této diplomové práci. V roce 2009 vyšla nová verze nazvaná NXT v2.0, která obsahuje navíc senzor rozpoznávající barvy a novou verzi softwaru a firmwaru. Při popisu NXT čerpáme z [9].

2.1 LEGO NXT

LEGO NXT kit se skládá z několika částí:

- NXT inteligentní kostka,
- NXT senzory,
- NXT komponenty pro stavbu robota (klasické LEGO součástky),
- software pro programovaní robota v jazyce NXT–G.

Inteligentní kostka obsahuje 32 bitový mikroprocesor s FLASH pamětí a 100x64 pixelů velký monochromatický displej. Pod displejem nalezneme čtyři tlačítka pro nastavení robota a spouštění programů. Můžeme je využít v programu a přiřadit jim nějakou funkci. Shora nalezneme tři výstupní porty a jeden USB 2.0 port, zdola čtyři vstupní porty. Napájení je možno buď 6 AA 1.5V bateriemi nebo Li-Ion baterií. Krom toho disponuje integrovaným reproduktorem, kterým je schopen přehrát zvukové soubory s vzorkovací frekvencí až 8kHz a Bluetooth v2.0. Inteligentní kostka je zobrazena viz Obrázek 1 na následující straně.

LEGO uvolnilo firmware pro NXT jako OpenSource, proto jsou k dispozici API pro různé jazyky. Většina programovacích jazyků pracuje s originálním firmwarem, některé používají jeho různé modifikace např. RobotC[7] (založeno na C). V této práci je použita nejnovější verze firmware 1.28.

Vstupní a výstupní porty jsou osazeny konektory RJ12 s šesti vodiči. Jsou podobné telefonním RJ11, ale vzájemně nekompatibilní. Firma LEGO možná chtěla zabránit připojení jiných zařízení, které by mohly kostku poškodit. Výstupní porty označené A, B, C slouží k připojení motorů. Motory jsou kvalitní krokové motory s nejmenším krokem 1°. Vstupní porty jsou označené čísly 1 – 4, ty slouží k připojení senzorů. V základním kitu nalezneme dotykový senzor, pomocí kterého můžeme například zaznamenat náraz robota na překážku. Dále pak světelný senzor, rozeznávající stupně šedi, mikrofon např.

Obrázek 1: NXT inteligentní kostka [9]

pro reakci robota na potlesk a nakonec ultrazvukový senzor pro měření vzdálenosti. Na první pohled se zdá, že k robotovi můžeme připojit jen 4 vstupní zařízení, zdání ale klame. Tyto porty podporují totiž sběrnici I^2C a RS-485. I^2C sběrnice umožňuje připojit až 127 zařízení v sérii. Tato sběrnice normálně využívá 4 vodiče – napájení, zem, hodinový signál a data. NXT používá 6 vodičů, protože vyvádí z kostky dvě různá napětí +9V jako analogový interface (kvůli zpětné kompatibility se starším RIS) a +4.3V. Takto můžeme připojit k robotovi cokoliv, pak jen stačí naprogramovat komunikaci s tímto zařízením.

Nedílnou součástí jsou mechanické komponenty pro sestavení robota. LEGO přikládá několik návodů pro různé konstrukce robota. Další konstrukce lze vytvořit zkombinováním více kitů a komponent rozšiřujících kit. Lze vytvořit prakticky cokoliv, od chodícího robota až po tiskárnu.

2.2 Senzory

Bez senzorů by byl robot NXT jen obyčejná hračka s programovatelným displejem. Senzory dělají z NXT něco víc, něco co může prozkoumávat prostor nebo reagovat na různé podněty. LEGO vytvořilo několik senzorů, seznam se neustále rozšiřuje. Kromě firmy LEGO vytvářejí senzory i jiné firmy, jako např. firma HiTechnic [2]. Díky komunikaci přes I^2C si své senzory vytvářejí i domácí kutilové a zveřejňují je na internetu. To dává podnět firmám, tvořícím tyto senzory, inspiraci k hromadné výrobě. Některé senzory LEGO dodává přímo s kitem, ostatní lze objednat přes internet. Ne všechny senzory jsem měl možnost otestovat.

2.2.1 NXT Touch Sensor

Asi jeden z nejzákladnějších senzorů obsahující pouze tlačítkový spínač. Může být použit jako detekce nárazu robota do nějaké překážky nebo jako kontrola uchycení míčku. Ze

dvou těchto senzorů lze udělat drátové dálkové ovládání (jedno tlačítko směr doleva nebo doprava, druhé směr dopředu nebo dozadu).

2.2.2 NXT Light Sensor

Základem světelného senzoru je fotocitlivá dioda. Ta umožňuje snímat osvětlení. Pro možnost testovaní odstínů povrchu předmětů obsahuje senzor ještě světlo vyzařující diodu LED (Light – Emitting Diode), která je umístěná hned vedle diody fotocitlivé. Pak senzor funguje následovně: LED vyzařuje světlo, to se odráží od snímaného objektu zpět do senzoru. Tím lze rozeznat stupně šedi barvy objektu. Záleží samozřejmě na povrchu objektu, jestli je matný nebo lesklý, plochý nebo členitý. Senzor dokáže pracovat ve dvou režimech:

1. aktivní – s rozsvícenou LED diodou,
2. pasivní – se zhaslou LED diodou.

Aplikací pro aktivní režim může být například rozpoznání povrchu sebraného tělesa nebo sledování čáry. Pasivní režim může být použit například pro změnu rychlosti jízdy robota v závislosti na osvětlení v místnosti.

2.2.3 NXT Sound Sensor

Senzor s mikrofonem měří intenzitu okolního zvuku v dB a dBA. Snímá frekvenci kolem 3 – 6kHz (lidské ucho slyší cca 0 – 16kHz). Může být použit například pro zastavení robota potleskem.

2.2.4 NXT RF (Radio frequency) ID Sensor

Tento senzor je novinkou z dílen LEGA. Umožňuje bezdrátově číst kódy čipů. Je dodáván se dvěma bezdrátovými čipy. Využití je libovolné, např. po přiložení prvního čipu proveď jednu rotaci kolem své osy, po přiložení druhého čipu proveď dvě rotace.

2.2.5 NXT Ultrasonic Sensor

Ultrazvukový senzor je asi jeden z nejužitečnějších senzorů. Umožňuje robotovi rozeznat vzdálenost od překážky v rozmezí od 0 – 250cm. Funguje na principu odrazu zvuku. Díky známé rychlosti šíření zvuku se zaznamená čas vyslání zvuku a čeká se, dokud se odražený zvuk nevrátí zpět. Pak se z rozdílu těchto dvou časů vypočte vzdálenost. Funguje přesně, ale záleží opět na povrchu tělesa a úhlu, který svírá senzor a snímaný objekt. Některé tělesa zvuk pohltí a neodrazí vůbec zpět, pak se zdá, že je těleso příliš daleko. Nebo naopak u těles s dobrou odrazivostí může dojít k odrazu mimo, pokud není těleso přibližně kolmo k snímači. Častou aplikací je prohledávání bludiště tvořeného zdmi nebo rozeznávání vzdálenosti.

2.2.6 HiTechnic EOPD (Electro – Optical Proximity Detector) Sensor

Tento senzor je určen stejně jako ultrazvukový senzor pro snímání vzdálenosti od objektu. Je založen na podobném principu jako světelný senzor, ale pracuje jen v aktivním režimu (pouze s rozsvícenou LED diodou). Fotocitlivá dioda je závislá pouze na světle speciální LED diody. Proto snímač funguje bez ohledu na okolní osvětlení. Tím je možno ho použít za různých světelných podmínek. Oproti ultrazvukovému senzoru je mnohonásobně citlivější, ale měří jen do vzdálenosti cca 20cm. Nevrací hodnotu v centimetrech, pouze jen bezrozměrné číslo v rozsahu 0 – 100 nebo jako nezpracovaná data 0 – 1023. Výhoda je, že funguje i pod většími úhly svíranými s objektem něž ultrazvukový senzor.

2.2.7 HiTechnic Acceleration Sensor

Senzor snímá pohyb v souřadnicích *x,y,z*. Použití: např. robot s tímto senzorem může odesílat informace o svém pohybu druhému robotovi, a ten bude kopírovat jeho pohyb.

2.2.8 HiTechnic Compass Sensor

Kompas senzor umožňuje robotovi snímat sever. Vrací hodnoty v rozmezí 0 – 359°, sever je 0°. Aby se omezily nežádoucí vlivy (motorů, baterií), musí být umístěn alespoň 10cm od kostky a 15cm od motorů. V okolí senzoru samozřejmě nesmí být žádné magnety. Může být použit pro otáčení robota na sever, nebo otáčení o určitý úhel. Rychlost snímání je přibližně 100krát za vteřinu. Senzor je možno taky kalibrovat pro snížení nežádoucích účinků magnetického rušení (motorů aj.).

2.2.9 HiTechnic Color Sensor

Dokáže detekovat více jak 15 barev, je nastaven na rozpoznávání standardních LEGO barev (např. barvy míčků v kitu aj.). Funguje na podobném principu jako světelný senzor, ale červená LED dioda byla nahrazena bílou LED diodou.

2.2.10 HiTechnic Gyro Sensor

Jedná se o malý gyroskop. Zaznamenává velikost změny úhlu za vteřinu v rozsahu +/- 360°. Použití může být například pro udržení rovnováhy robota na dvou kolech. Rychlost snímání je přibližně 300krát za vteřinu.

2.2.11 HiTechnic IRLink Sensor

Jde o infračervený modul, jaký je možno nalézt v některých mobilech. Pomocí něj může NXT ovládat rychlost a směr některých LEGO zařízení podporující IR komunikaci např. LEGO vláčky(7897 a 7898), buldozer (8275) a Monster Dino (4958)[3].

2.2.12 HiTechnic IRReceiver Sensor

Tento senzor je určen pro příjem a dekódování IR signálů z dálkového ovládání LEGO (8879 a 8885). Pomocí dálkového ovládání pak lze ovládat směr a rychlost robota, popřípadě jinou funkci programu.

2.2.13 HiTechnic IRSeeker V2 Sensor

Tento senzor umožňuje detekovat IR signály standardních dálkových ovladačů nebo HiTechnic IRBallu a určit jejich sílu a směr, ze kterého vycházejí. Realizovat tak lze fotbal robotů nebo robota, který Vás sleduje, když mačkáte tlačítko ovladače od televize.

2.3 LEGO Mindstorms NXT (NXT–G)

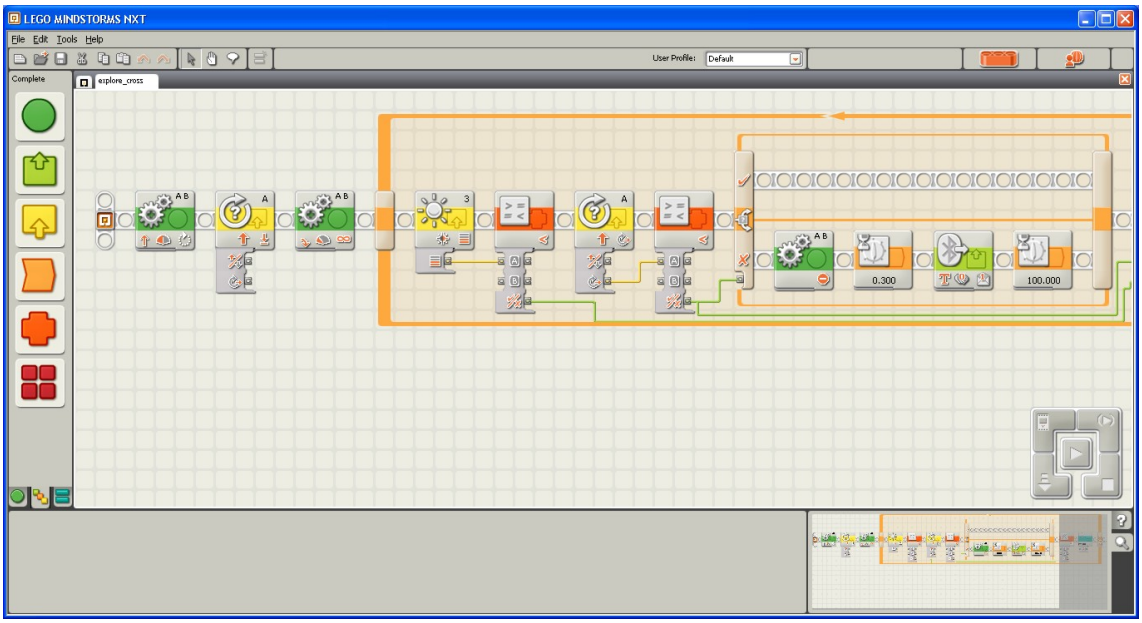
LEGO přikládá ke kitu vývojové prostředí na programování robotů, které vytváří programy jazykem NXT–G. Jedná se o grafický, snadno pochopitelný jazyk kompilovaný přímo do FLASH paměti kostky. Po kompilaci se spouští přímo na kostce pomocí tlačítek. Základem grafického programování je skládání existujících bloků systémem *drag and drop* (táhni a pust') dohromady. LEGO připravilo několik základních bloků, které postačují pro vytvoření jednoduchých programů. Mezi bloky nalezneme bloky pro:

- všechny dodávané senzory LEGO a ovládání motorů,
- řízení toku programu (cykly, podmínky),
- zápis na displej, konverzi čísla na text,
- základní matematické operace a proměnné.

Na LEGO [4] stránkách jsou k dispozici nové verze bloků. Funkci mají stejnou jako předchozí bloky, ale jsou optimalizovány tak, že po kompilaci nezabírají moc místa ve FLASH paměti NXT. Krom oficiálních LEGO bloků lze stáhnout i jiné bloky např. pro senzory vyráběné jinými firmami, bloky pro komunikaci přes I^2C aj. Obrázek 2 na následující straně zobrazuje vývojové prostředí.

Bloky lze libovolně kombinovat a seskupením několika bloků můžeme vytvořit vlastní blok, který později znovu použijeme. To provedeme tak, že kurzorem vybereme bloky, ze kterých chceme vytvořit vlastní blok. V menu zvolíme *Edit* → *Make A New My Block*. Používání vlastních bloků šetří pamět'. Pokud v jednom schématu použijeme více stejných bloků, zabere pamět' pouze jeden. Pokud však ve schématu blok použijeme jen jednou, zabere víc místa v paměti, než kdyby bylo schéma tvořeno jen základními bloky. Bloky se provádějí postupně tak, jak jsou poskládány za sebou. Lze ale vytvořit tři paralelně běžící větve. Bloky se napojují rovnoběžně a vytvářejí tak přehledné schéma. Některé bloky se musí propojit pomocí „drátových“ propojek (Obrázek 4 na straně 9) pro přenos hodnot proměnných.

Mezi velké nedostatky patří nemožnost rekurze bloků. Pokud např. chceme v jednom bloku volat stejný blok, musíme vytvořit kopii bloku a přejmenovat ji. To ovšem neřeší možnost rekurzivního volání stejného bloku, pouze dává možnost volat znova tentýž blok



Obrázek 2: LEGO Mindstorms NXT (NXT–G) vývojové prostředí

pod jiným názvem. Dalším nedostatkem je to, že při vytvoření většího schématu se stane vývojové prostředí naprosto neovladatelné. Potom je problém s bloky manipulovat a přetáhnout někam jinam. Řešením je velké schéma rozdělit pomocí vlastní bloků do několika menších. Dalším problémem byl přechod na novější verzi 2.0. Při něm se smazaly definice proměnných z bloků nadřazených. Proměnné se pak chovaly jako znova nadefinované a nepřenesly hodnotu uloženou v bloku o úroveň výš.

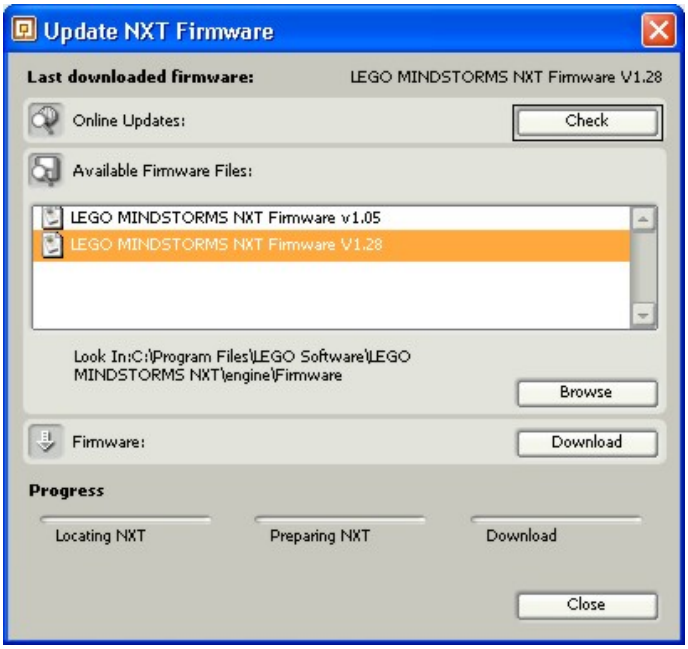
2.3.1 Seznámení s vývojovým prostředím

Po spuštění vývojového prostředí LEGO Mindstorms NXT se nám objeví úvodní obrazovka, na které si můžeme vybrat, jestli chceme založit nový projekt nebo otevřít starší. Krom toho LEGO na této obrazovce nabízí prohlédnutí dvou videí pro seznámení s prostředím a uživatelský manuál. Vřele doporučuji tato videa shlédnout. Zvolíme vytvoření nového projektu. Uprostřed se nám objeví prázdné plátno, na které můžeme dávat bloky. Bloky nalezneme vlevo od plátna, jsou rozděleny do třech záložek. První obsahuje běžně používané bloky, druhá kompletní seznam bloků a třetí uživatelské bloky a bloky stažené z internetu. Horní lišta obsahuje ikony pro standardní práci s dokumentem, pak ikony pro přepínaní režimu kurzoru – výběr nebo posuv. Vpravo nalezneme ikonu, která ukrývá návody na sestavení různých tvarů robota a ikonu s novinkami na internetu. Na plátně jsou ikony pro stažení programu do robota, spuštění nebo zastavení programu a informace o kostce. Pod plátnem je náhled na celé schéma a nápovědu. Menu nahoře obsahuje klasické položky jako *File*, dále *Edit* – zde jsou položky pro vytváření a editaci vlastních bloků, vytváření vlastních proměnných. V menu *Tools* nalezneme položky pro kalibraci

senzorů, aktualizaci firmware, vytváření vlastních balíků, import a export bloků, editor obrázků a zvuku pro NXT a dálkové ovládání robota.

2.3.2 Instalace firmware

Nejprve je důležité nalézt nejnovější firmware, ten je k dispozici na LEGO[4] stránkách v sekci *Support* → *Files*. Nový firmware může opravovat různé chyby a přinášet nové funkce. Po stažení firmwaru do počítače zvolíme *Tools* → *Update NXT Firmware*. Objeví se okno, viz Obrázek 3, kde vybereme cestu k firmwaru a stiskneme tlačítko download.

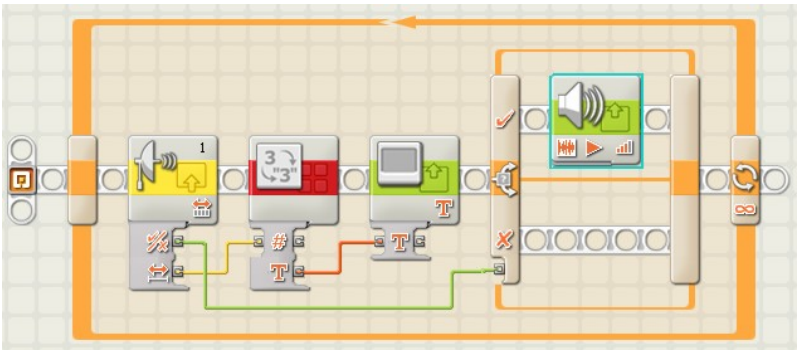


Obrázek 3: LEGO Mindstorms NXT – Update NXT Firmware

2.3.3 Tvorba programu pro robota

Vytvoříme jednoduchý program na zobrazení vzdálenosti ultrazvukového senzoru na displeji a detekci překážky. Pokud bude vzdálenost menší než 20cm, spustí se výstražný zvuk. Nejprve připojíme k robotovi ultrazvukový senzor k portu č.1. Program musí běžet ve smyčce, jinak by robot získal hodnotu ze senzoru a program by se ukončil. Proto vložíme do schématu cyklus – klikneme na záložku *Complete Palette*, tam vybereme *Flows* → *Loop* a přetáhneme do schématu. Přidáme ultrazvukový senzor *Sensor* → *Ultrasonic Sensor*. Klikneme na senzor a nastavíme ho na port 1, nastavíme jednotky na *cm*, vzdálenost na menší než 20. Blok ultrazvukový senzor má porovnávající blok přímo v sobě, takže nemusíme vkládat do schématu porovnávající blok a můžeme využít výstupu porovnání. Klikneme proto na spodní část bloku, tím se rozbalí vstupy a výstupy bloku. Vložíme konverzi čísla na text *Advanced* → *Number to Text* a displej *Action* → *Display*. Propojíme

výstup *Distance* se vstupem *Number* konvertoru, obdobně výstup *Text* se vstupem *Text* displeje. Tím se nám bude zobrazovat hodnota vzdálenosti na displeji. Přehrání zvuku je závislé na podmínce (vzdálenost je menší než 20cm), vložíme do schématu tedy podmínku *Flow* → *Switch*. Klikneme na blok podmínky a změníme *Control* na *Value – Logic*. Propojíme výstup bloku ultrazvukového senzoru *Yes/No* s vstupem podmínky. Do podmínky vložíme zvukový blok *Action* → *Sound* a nastavíme zvuk na *Object detected*. Výsledek by měl vypadat přibližně jako Obrázek 4. Tím je program hotov. Stačí propojit robota a stisknout na plátně tlačítko *Download and run*.



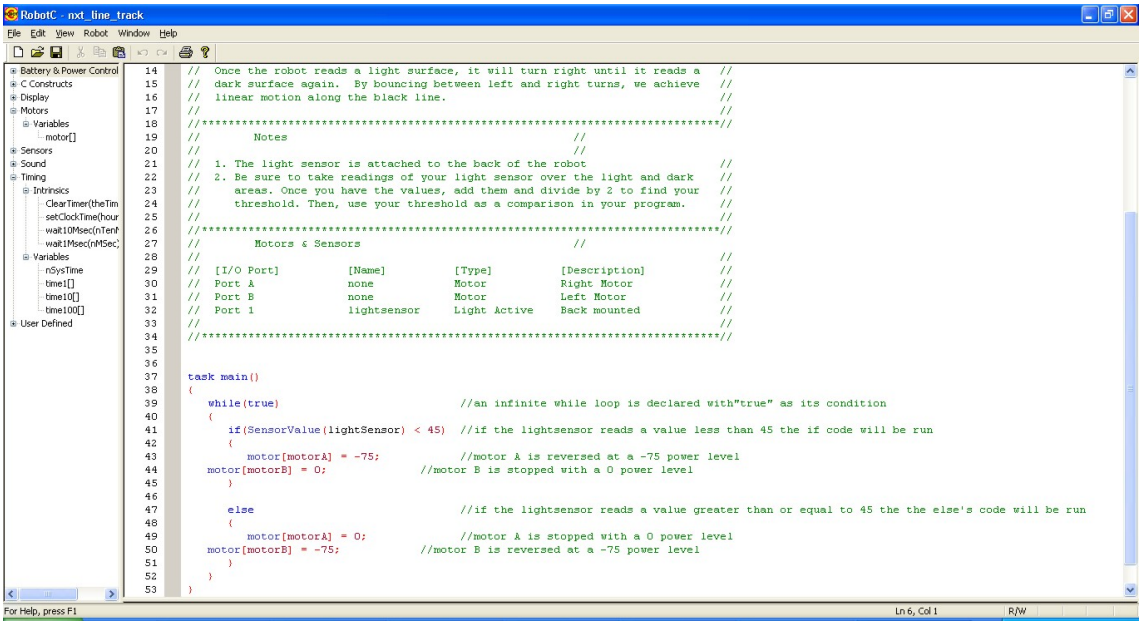
Obrázek 4: LEGO Mindstorms NXT – Detekce objektu

2.3.4 Zhodnocení

Jazyk NXT–G je snadno pochopitelný, a proto je to určitě nejlepší volba pro začátečníky v programování robotů NXT. Navíc je přímo od výrobce, tím by mělo být zaručeno, že nedojde k poškození NXT např. chybou v cizím firmware. Protože se program stahuje přímo do paměti robota, není potřeba k spouštění programů počítač a reakce na stav senzorů je rychlejší. Nevýhodou je absence debuggeru. Bez něj by byl pro komplikovaný projekt tento nástroj možná nedostačující.

3 RobotC

RobotC[7] je vývojové prostředí a programovací jazyk (založený na jazyku C) pro programování robotů LEGO Mindstorms NXT, RCX a Innovation FIRST VEX (pro každou platformu existuje jiný instalační balík). Vytvořila jej univerzita Carnegie Mellon, 30 denní demoverze je k stažení na domovských stránkách. Stejně jako NXT–G nahrává své programy přímo do paměti NXT, k jejich spuštění je ale potřeba nahrát RobotC firmware. Tento firmware ovšem přináší mnoho výhod oproti originálnímu firmware. Mezi velké přednosti patří větší rychlost provádění operací, a tím rychlejší čtení dat ze senzorů, více rozhodnutí nebo výpočtů za sekundu. Dále je rychlejší i posílání zpráv mezi PC a NXT, udává se o 50 – 500% oproti originálnímu firmwaru. Jedinou nevýhodou je nekompatibilita s MRDS. Prostředí obsahuje taky interaktivní *real-time debugger*, který dělá z tohoto prostředí opravdu kvalitní nástroj pro programování robotů. K dispozici jsou také připravené ukázkové úlohy a rychlý průvodce s nápovědnými videy. Obrázek 5 ukazuje vývojové prostředí RobotC.



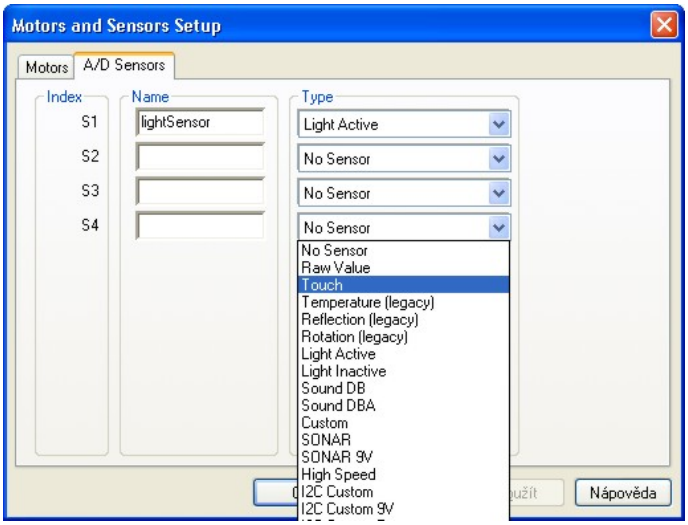
Obrázek 5: RobotC vývojové prostředí

3.1 Seznámení s vývojovým prostředím

V době psaní této práce jsem měl k dispozici vývojové prostředí ve verzi 1.05. Po spuštění prostředí RobotC se objeví okno rozdělené do dvou částí. Pravá část je určena pro psaní kódu tak, jak jsme zvyklí z ostatních vývojových prostředí (IDE). Levá část je tvořena knihovními RobotC API funkcemi, které můžeme systémem *drag and drop* přetáhnout do kódu. Kód se skládá obdobně jako kód jazyka C z hlavní spouštěcí metody *main*. V RobotC začíná každá úloha klíčovým slovem *task*. První řádky označené jako *auto* jsou

tvořeny nastavením portů, ty se ale needitují přímo v editoru, nýbrž ve speciálním okně k tomu určeném. Pak se v případě změny automaticky přegenerují.

Nahoře v menu *File* nalezneme kromě klasických položek položku pro otevření ukázkových příkladů a otevření s kompilací. V menu *Edit* nalezneme navíc kopírování konfigurace připojených portů do schránky. Další menu položky jsou závislé na režimu zobrazení. Na výběr máme *Basic* nebo *Expert*. Nastavení nalezneme v menu *Window* → *Menu Level*. Budeme popisovat *expertní* režim. Následuje položka *View*. Tady nalezneme položky pro zobrazení a smazání logu zpráv, nastavení editoru a rychlé nastavení editoru do dvou profilů – *debug* a *release*. Další položka je pro smazání všech hodnot z registrů. Následující položky slouží k zobrazení a skrytí panelů. Asi nejdůležitější položkou menu je položka *Robot*. V ní se ukrývají základní funkce vývojového prostředí jako např. kompilace a stáhnutí programu do našeho NXT. Je zde i položka pro stažení firmware, doporučuji ale použít originální LEGO software pro stažení firmwaru, viz kapitola 2.3.3. Občas se totiž stalo, že se firmware stahoval příliš dlouho nebo se stahovaní nedokončilo. V menu nalezneme i debugger, který poskytuje krokování programu. V položce *Robot* → *NXT Brick* nalezneme nástroje pro správu souborů v NXT, test I^2C sběrnice, nastavení připojení NXT a nástroj s informacemi o aktuálním stavu senzorů a motorů. Dále následuje položka pro nastavení robotické platformy RCX nebo NXT. Jako další je položka pro nastavení motorů a senzorů viz Obrázek 6. Zde nastavíme připojené senzory a motory, potvzením tlačítka *Ok* si necháme vygenerovat konfiguraci do zdrojového kódu.



Obrázek 6: RobotC vývojové prostředí – nastavení motorů a senzorů

3.2 Tvorba programu pro robota

Nejprve je potřeba nahrát RobotC firmware z instalační složky RobotC/firmware, buď přes originální software viz kapitola 2.3.3, anebo přes menu *Robot* → *Download Firmware*. Pro první program doporučuji použít jako vzor nějaký z ukázkových příkladů. Zvolíme

proto v menu *File* → *Open Sample Program*, tam otevřeme složku *Sonar* a v ní program *nxt_obstacle_detect*. RobotC nedovolí editaci ukázkových příkladů, proto si uložíme kopii programu někam jinam – *File* → *Save as ...*. Musíme nastavit porty, kde máme připojeny motory a ultrazvukový senzor. To provedeme v menu *Robot* → *Motors and Sensors Setup*. Zdrojový kód programu viz Výpis 1.

```
task main()
{
    do{
        motor[motorA] = 75;
        motor[motorB] = 75;
    }while(SensorValue(sonarSensor) > 20);
}
```

Výpis 1: RobotC – program pro detekci překážky

Ukázkový program funguje následovně. Nejprve se spustí *task main*. V něm je cyklus typu *do – while*, nejprve se provede blok *do*. V něm je nastavení rychlosti obou motorů na 75%. Pak dojde k porovnání hodnot z ultrazvukového senzoru se zadanou hodnotou (hodnota je v palcích). Pokud podmínka platí, cyklus se opakuje. V opačném případě dojde k ukončení cyklu, a tím k ukončení programu a robot se zastaví.

My bychom chtěli, aby robot na displej vypsal *Obstacle* (překážka), zahrál nějaký výstražný zvuk a zastavil se. Po odstranění překážky by se opět rozjel. Z původního cyklu uděláme klasický *while* cyklus a vložíme celý cyklus do nového nekonečného cyklu *while(true)*. Pokud bude blízko překážky, chceme aby se zastavil, proto vložíme za cyklus testující vzdálenost, nastavení rychlosti obou motorů na 0. Pak chceme, aby se na displeji zobrazilo *Obstacle*. Z levého menu s knihovními funkcemi vložíme *Display* → *Intrinsics* → *nxtDisplayStringAt*. Nakonec chceme zahrát výstražný zvuk. Obdobně z levého menu vybereme *Sound* → *Intrinsics* → *playSoundFile*. Zvukové soubory k přehrání nalezneme v menu *Robot* → *NXT Brick* → *File Management*. Ještě je potřeba do původního cyklu vložit vymazání displeje – *Display* → *Intrinsics* → *eraseDisplay*. Zdrojový kód programu viz Výpis 2.

```
task main()
{
    while(true){
      while(SensorValue(sonarSensor) > 20){
          eraseDisplay();
          motor[motorA] = 75;
          motor[motorB] = 75;
      }
      motor[motorA] = 0;
      motor[motorB] = 0;
      nxtDisplayStringAt(7, 7, "Obstacle");
      PlaySoundFile("Obstacle_detected.rso");
    }
}
```

Výpis 2: RobotC – upravený program pro detekci překážky

Nakonec stačí program zkompilovat a nahrát do FLASH paměti robota. To provedeme přes menu *Robot* → *Compile and Download Program*.

3.3 Zhodnocení

Jedná se o velice propracovaný nástroj, ve kterém jdou řešit komplikované projekty. Podobnost programovacímu jazyku C je velkou výhodou. Díky znalosti jazyka C a spousty ukázkových příkladů se lze v RobotC rychle naučit programovat. Knihovní funkce neznají mezí a lze si naprogramovat I^2C komunikaci s prakticky libovolným, i vlastnoručně vyrobeným zařízením. Díky debuggeru a propracovaným nástrojům na sledování stavu senzorů a motorů máme naprostou kontrolu nad tím, co se aktuálně provádí. Výhodou může, ale nemusí být, optimalizovaný firmware (díky nekompatibilitě s MRDS). Celkově hodnotím RobotC jako nejlepší nástroj k programovaní NXT, který jsem měl možnost poznat.

4 Microsoft Robotics Developer Studio (MRDS)

Microsoft přišel s revoluční myšlenkou – oprostit se od hardware, pro který jsou IDE pro robotická zařízení psána a s pomocí služeb vytvořit nástroj nezávislý platformě. V podstatě jde o to, že pro každý hardware se napíše služba poskytující přístup ke svým portům, se kterými pak ostatní služby komunikují. Sjednocení vývojového prostředí pro více platforem mělo jediný cíl – a to úsporu času a peněz znovupoužitím existujících knihoven a jednotný přístup ke všem robotickým zařízením. Následující informace jsou čerpány z [8, 10, 11]. Hlavními požadavky pro vznik této technologie byly:

- Snadné nastavení senzorů a motorů, možnost je spouštět asynchronně.
- Spouštění a zastavování softwarových komponent dynamicky.
- Interaktivní sledování robota a jeho řízení.
- Dovolení více uživatelům sdílet přístup k jednomu robotovi nebo jednomu uživateli povolit přístup k více robotům.
- Znovupoužití softwarových komponent napříč robotickými platformami.

Microsoft dodává vývojové prostředí v několika verzích. Zdarma je k dispozici *Express* verze, která ale má různá omezení ve všech nástrojích. Např. z VPL nelze generovat do C# kódu, není podpora .NET Compact Frameworku a další omezení. Více informací nalezneme v [5]. Pro komerční účely je dostupná placená verze *Standard* v ceně 500$. Microsoft uvolnil pro akademické účely zdarma verzi *Academic* bez omezení, kterou lze stáhnout přes MSDN Academic Alliance nebo DreamSpark^{TM}. MRDS obsahuje nástroje:

- Visual Programming Language (VPL),
- Visual Simulation Environment (VSE),
- DSS Manifest Editor (DSSME),
- DSS Log Analyzer,
- a další.

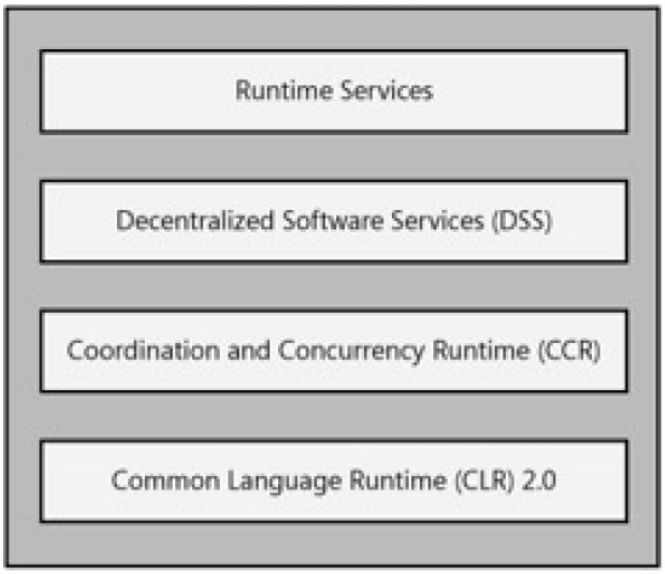
VPL je grafický nástroj na programování robotických aplikací. Nástroj VSE slouží k simulaci a testování robotických aplikací ve 3D s podporou fyziky. DSS Manifest Editor slouží pro vytváření a editaci konfigurací aplikace. DSS Log Analyzer, jak název napovídá, slouží k analýze toku zpráv mezi službami s možností detailního zobrazení zprávy.

Krom nástrojů přikládá Microsoft k MRDS spoustu ukázkových příkladů a tutoriálů. Příklady popisují jak vytvářet služby, dále jsou to příklady pro seznámení s nástroji VPL a VSE a ukázky služeb pro platformy jako je Fischertechnik, iRobot, Kondo, LEGO a MobileRobots.

4.1 Architektura

MRDS je založeno na .NET, proto pro programování můžeme využít libovolný z jazyků podporovaných v .NET (C++, C#, Visual Basic .NET, Dephi, Python aj.), ale většina ukázkových příkladů je psaná v C# a Visual Basic .NET. Navíc VPL umí schéma přegenerovat pouze do C# kódu.

Běhové prostředí MRDS je založeno na Common Language Runtime (CLR). Využívá dvou klíčových prvků Concurrency and Coordination Runtime (CCR) a Decentralized Software Services (DSS) k vytvoření množiny podporující centrální objekt MRDS – služby. Obrázek 7 ukazuje běhové prostředí MRDS. CLR poskytuje oběma knihovnám přístup k .NET Frameworku.

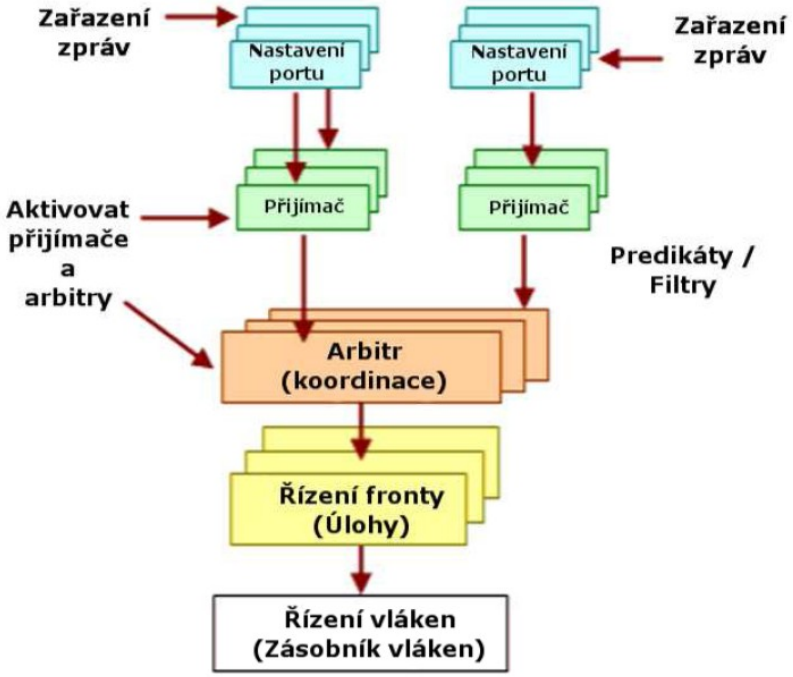


Obrázek 7: MRDS běhové prostředí [11]

4.1.1 Concurrency and Coordination Runtime (CCR)

CCR je hlavní knihovna MRDS starající se o koordinaci jednotlivých aktivit (např. informace ze snímačů, pohyb robota aj.). Poskytuje asynchronní běh těchto aktivit pomocí vláken (např. během otáčení robota se provádí čtení hodnot nějakého senzoru). CCR je .NET knihovna, kterou může využít libovolná .NET aplikace vyžadující asynchronní zpracování. Knihovna zakrývá práci s vlákny a tím zjednodušuje programování aplikací s asynchronním zpracováním. Základem knihovny jsou porty reprezentované třídou *Port*. Port může být libovolný snímač nebo motor robota. Každý port si udržuje frontu zpráv nebo dat určitého typu. Ze senzorů aplikace data čte a do motorů je zapisuje. Schéma CCR viz Obrázek 8 na následující straně.

Další důležitou třídou je třída *Arbiter*. Tato třída umožňuje vytvářet handlery na události portů (pokud přijde zpráva do nějakého portu). S pomocí této třídy se dají události



Obrázek 8: MRDS architektura CCR [10]

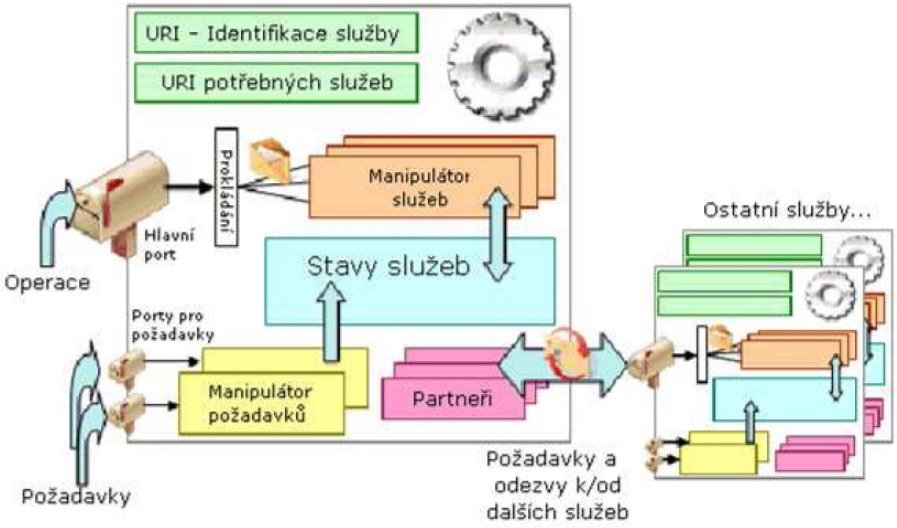
propojovat, a vytvářet tak nějaké podmíněné události (např. proveď nějakou akci pokud přijdou zprávy na tyto dva porty). Toto propojení se v CCR nazývá *join*. Příkladem *joinu* může být jízda robota s ultrazvukovým a světelným senzorem. Pokud bude vzdálenost překážky menší jak 20cm a zároveň světelný senzor bude snímat černou barvu, pak se robot zastaví. Dalším propojením je propojení typu *choice*. To naopak čeká na příjem zprávy z libovolného portu ze skupiny vybraných portů, a pak provede určitou akci. Tak jako v předchozím příkladě, tentokrát by robotovi k zastavení stačila buď vzdálenost od překážky menší jak 20cm, anebo sejmutí černé barvy světelným senzorem. Každý handler vrací návratovou hodnotu *IEnumerator<ccr.ITask>*, díky tomu může postupně asynchronně vracet příkazy, které se mají v rámci ošetření události vykonat (vrací se pomocí klíčového slova *yield return*).

4.1.2 Decentralized Software Services (DSS)

DSS je aplikační model, který umožňuje vývojářům interaktivně sledovat služby v reálném čase. Je založen na REST (Representational State Transfer). REST je architektura rozhraní pro webové aplikace, navržená pro distribuované prostředí. V prostředí WWW (World Wide Web) se při každém novém požadavku na zobrazení stránky ztrácí informace o předešlém požadavku. Vývojáři .NET sice mohou použít *session* proměnné pro uložení stavu mezi požadavky, ale každý požadavek je nezávislý na dalším požadavku. REST umožňuje a definuje efektivní způsob přístupu k těmto datům. Využívá Hypertext

Transfer Protocol (HTTP) a skrz něj pomocí Extensible Markup Language (XML) vrací rychle strukturovaná XML data. Díky tomu můžeme ovládat a sledovat roboty vzdáleně a efektivně.

DSS umožňuje službám komunikovat s jinými službami nezávisle na tom, kde služba běží. Každá služba má jednoznačný identifikátor Uniform Resorce Identifier (URI) v rámci MRDS. Každá služba, která komunikuje s jinými partnerskými službami, zná jejich URI. Zjednodušené schéma služeb je zobrazeno viz Obrázek 9.



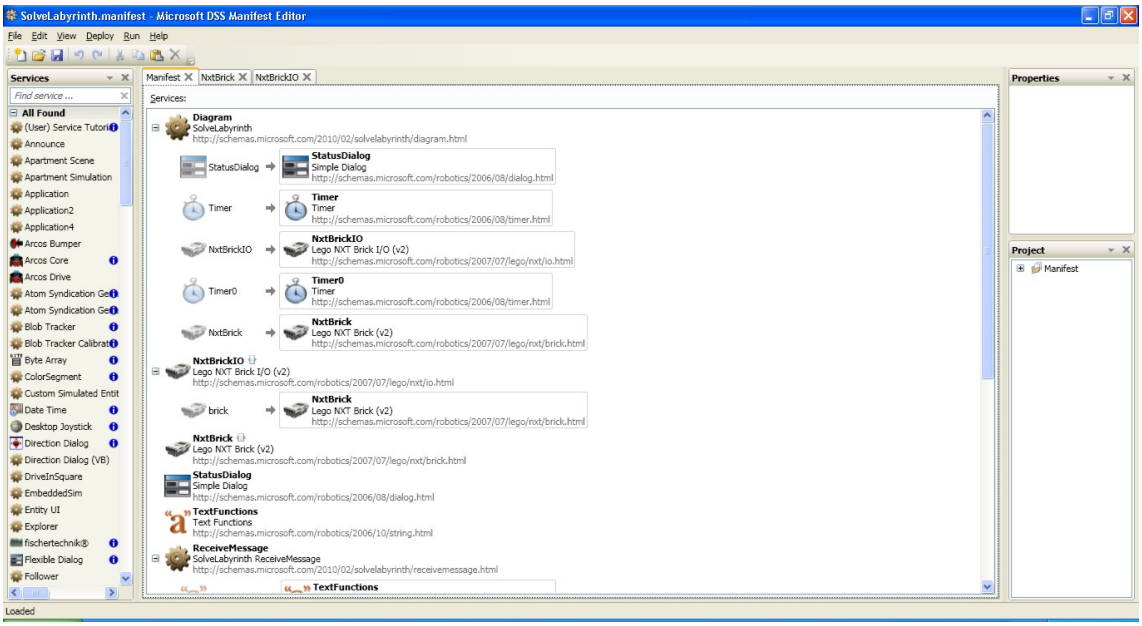
Obrázek 9: MRDS architektura služeb [10]

4.2 DSS Manifest Editor (DSSME)

Soubory typu *Manifest* (přípona .Manifest.xml nebo .xml) lze sice editovat v libovolném textovém editoru, jsou to pouhé textové soubory v XML. Je ale potřeba znát přesný formát jednotlivých položek. Proto je vhodnější použít DSSME viz Obrázek 10 na následující straně. Pokud používáme k vytváření aplikace nástroj VPL, není použití DSSME nutné. VPL za nás vygeneruje potřebný manifest a poskytuje grafické rozhraní k editaci manifestů partnerských služeb.

Na levé straně je rolovací seznam se všemi dostupnými službami. Služby se načítají z adresáře *bin*. Pokud chceme přidat do manifestu svoji službu, musíme ji do tohoto adresáře nahrát a pak v menu zvolit *View* → *Reload Services*. Služby lze jednoduše přidávat do manifestu systémem *drag and drop*, uprostřed pak vidíme celý diagram manifestu se všemi vloženými službami. Pokud na některou klikneme, uvidíme vpravo její vlastnosti, které můžeme snadno editovat.

Krom editace manifestů umí DSSME vygenerovat samorozbalovací archív celé aplikace na základě jejího manifestu. To lze jednoduše provést v menu *Deploy* → *Create Deployment Package*.



Obrázek 10: MRDS DSSME

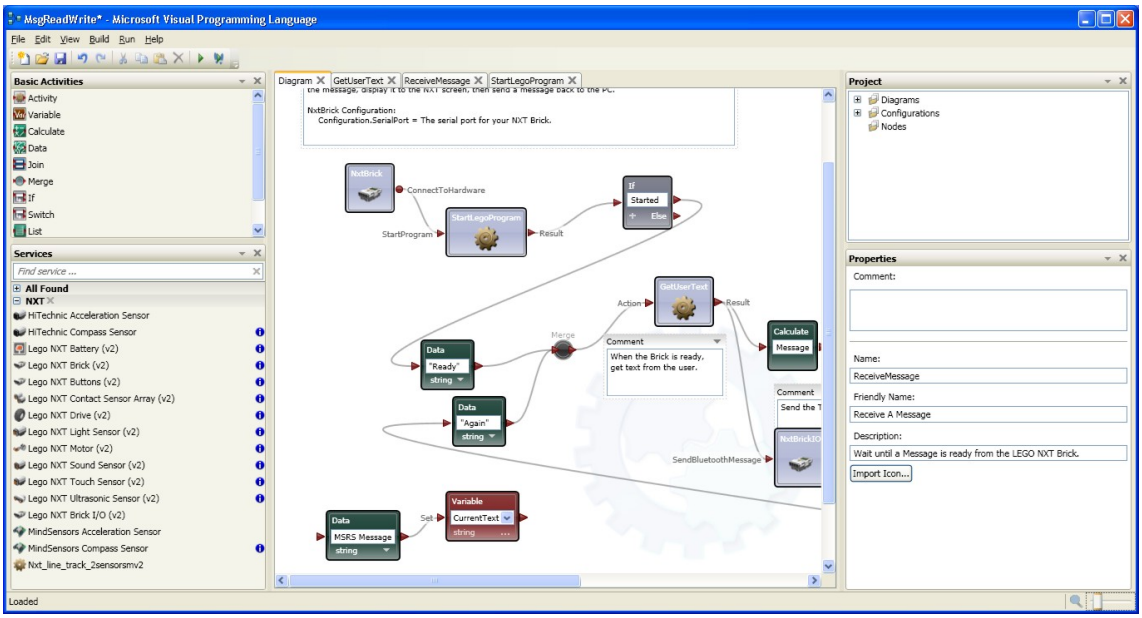
4.3 Visual Programming Language (VPL)

Je to grafický programovací nástroj založený na systému *drag and drop*, viz Obrázek 11 na následující straně. Je určen pro začínající vývojáře v MRDS. Oproti NXT–G umožňuje rekurzi, a tím mohou být programy využívající stejné funkce jednodušší.

4.3.1 Seznámení s vývojovým prostředím

Nástroj VPL je dosti podobný DSSME, jen uprostřed místo manifestu máme diagram aplikace v záložce, a vlevo je krom rolovacího seznamu služeb seznam bloků základních aktivit. Aktivitami se rozumí prvky programování, jejichž výčet je následující: *Activity, Variable, Calculate, Data, Join, Merge, If, Switch, List, List Functions, Comment.*

Pomocí bloku *Activity* můžeme vytvářet nové služby z existujících služeb a z bloků základních aktivit. Po přetažení bloku *Activity* do diagramu dvojklikem otevřeme nový diagram (jako novou záložku) této aktivity. Aktivita má vstup pro příjem dat, výstup pro odeslání výsledku a výstup pro notifikaci o nějaké události. Každá aktivita se může skládat z několika akcí. Ke každé akci definujeme typ vstupu a výstupu přes ikonu nahoře nad diagramem. *Variable* slouží k ukládání a načítaní proměnných. Pro jednoduché matematické operace (jako $+, -, *, /, \%$) využijeme blok *Calculate*, v případě složitějších operací je k dispozici služba *Math Functions*. Pomocí bloku *Data* nastavujeme hodnotu proměnných jako vstup do bloku. Blokem *Join* čekáme, dokud nepříjdou data ze všech připojených vstupů. *Merge* nám poslouží k vytváření cyklů. Spojuje datové toky do jednoho. Následuje blok *If*, výsledkem každé podmínky musí být hodnota *True* nebo *False*. Podobným blokem je *Switch*, ten oproti bloku *If* nemusí mít výsledek každé podmínky



Obrázek 11: MRDS VPL

True nebo *False*, ale musí být porovnatelný se vstupem. Pokud budeme potřebovat data ukládat do seznamu, můžeme použít blok *List*. K němu existuje blok *List Functions*, který obsahuje základní operace se seznamem (jako přídání prvku na konec a na určitý index, smazání prvku, spojení dvou seznamů, obrácení pořadí prvků, setřízení prvků, zjištění indexu prvku). Posledním blokem je blok *Comment*, pomocí něj vytvoříme textové pole, do kterého můžeme vložit komentář k diagramu.

Menu *File* disponuje navíc možností přidání diagramu do otevřeného projektu, ten se zobrazí jako nová záložka. V menu *Edit* se ukrývají funkce jako editace akcí a notifikací, proměnných, přípojení, datové připojení, ale ty jsou aktivní jen v určitých případech. Někdy je potřeba znovu načíst služby, to nalezneme v menu *View*. Tam jsou také užitečné položky pro zobrazení mřížky a zarovnávání bloků horizontálně či vertikálně. Jediná položka se nachází v menu *Build*, a to kompilace do C#. Pro spuštění můžeme použít menu *Run*, jiná možnost je použití příkazové řádky MRDS, nebo pokud máme diagram zkompilovaný jako službu, můžeme využít Visual Studio. V menu *Run* nalezneme spuštění programu, spuštění programu jako kompilované služby (pokud byl diagram pozměněn nebo došlo k překompilování ve Visual Studiu, budeme vyzváni k opětovné kompilaci). Dále je možnost spuštění v debug módu nebo spuštění distribuovaně na více počítačích. Zde nalezneme i nastavení portu (výchozí hodnota je 50000 pro HTTP a 50001 pro TCP), pod kterým služba běží a je možno zobrazit ji v prohlížeči na URL http://localhost:ČÍSLO_PORTU_HTTP/.

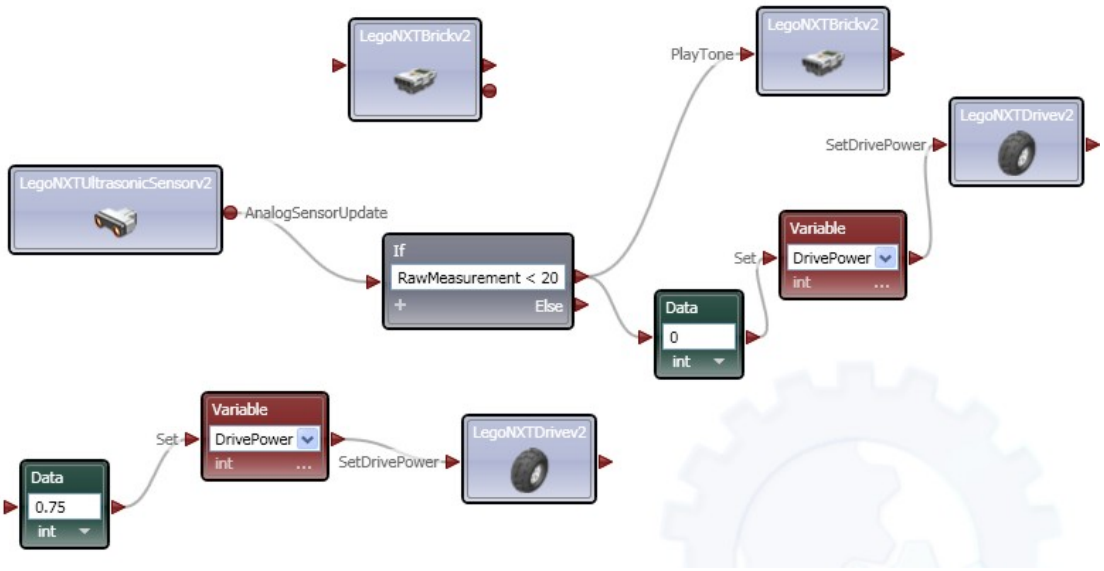
4.3.2 Tvorba programu pro robota

Opět jako v kapitole 3.2 a v kapitole 2.3.3 budeme vytvářet program pro detekci překážky. Zde si budeme muset vystačit se zahráním tónu po přiblížení k překážce (zobrazení textu na displej a přehrání zvukového souboru není k dispozici).

Po otevření nového projektu vložíme do diagramu službu *LEGO NXT Brick* (můžeme do vyhledávače zadat NXT), klikneme na ni a ve vlastnostech nastavíme číslo portu (Set initial configuration), se kterým máme kostku spárovanou. Poté vložíme službu *LEGO NXT Drive*, ta nám zajistí že robot pojede dopředu. Je potřeba u ní nastavit, ke kterým portům jsou motory připojeny (pro správné otáčení také rozteč a velikost kol). Je potřeba nastavit rychlost motorů, to provedeme vložením aktivity *Data* a *Variable*. *Data* nastavíme na hodnotu 0.75, kliknutím na *Variable* vytvoříme novou proměnnou *DrivePower*, typu *int*. Propojíme blok *Data* s blokem *Variable* a nastavíme *Set*. Pak propojíme blok *Variable* s blokem *LEGO NXT Drive* a nastavíme *SetDrivePower* na hodnotu *value.DrivePower* (naše proměnná). Můžeme zkusit spustit program v menu *Run* → *Start*, robot by měl jet bez zastavení (proto ho chyťte než do něčeho narazí). Teď je potřeba udělat detekci překážky, to provedeme vložením služby *LEGO NXT Ultrasonic Sensor* a nastavíme ji na port, kde je připojen ultrazvukový senzor. Vložíme aktivitu *If* a propojíme ultrazvukový senzor s touto aktivitou. Nastavíme *AnalogSensorUpdate* a v bloku *If* nastavíme *RawMeasurement* $<$ *20*. Označíme bloky *LEGO NXT Drive*, *Data* a *Variable* a vložíme znova jejich kopii (stačí CTRL+C a CTRL+V na vložené službě). Spojíme blok *If* se vstupem *Data* a nastavíme data na 0. Uděláme kopii *LEGO NXT Brick* a spojíme *If* s tímto blokem, nastavíme PlayTone, hodnoty nastavíme ručně (Edit values directly) na 1000ms pro trvání a 1000Hz pro frekvenci. Obrázek 12 na následující straně ukazuje jak by měl vypadat výsledek. Nakonec dáme program zkompilovat jako službu *Build* → *Compile as Service* a spustíme *Run* → *Compiled Services*. Můžeme si prohlédnout zkompilovanou službu ve Visual Studiu a upravit ji podle libosti, ale pozor – není možné z upravené služby vygenerovat zpět VPL diagram. Proto doporučuji při tvorbě programů udělat pokud možno co nejvíce funkcí ve VPL, a funkce, které se komplikovaně vytváří ve VPL, pak dopsat v C#. Při tvorbě programů pro robota se můžeme nechat inspirovat připravenými ukázkami diagramů od Microsoftu, které nalezneme v $MRDS\backslash samples\backslash VplExamples\backslash$.

4.3.3 Zhodnocení

VPL je sice určen pro začátečníky, ale dle mého názoru je komplikovanější než NXT–G. Služby totiž běží ve vláknech, a ne vždy je zcela jasné, kdy se která provede. V NXT–G se bloky skládají za sebe a je přesně dáno, kdy se provede určitý blok. Zde bloky sice lze zarovnat do roviny, ale propojení mezi bloky je tvořeno křivkou a složitější diagramy pak vypadají trochu nepřehledně. Další nevýhodou je nemožnost kompilace přímo do paměti robota, to ale přináší výhodu možnosti využití dokonalejších knihoven a spuštění distribuovaně na více počítačích. Bohužel právě tato vlastnost způsobuje u NXT nemožnost čtení hodnot ze senzorů v reálném čase. Velkou výhodou je možnost kompilace do strukturovaného jazyka C#, to zatím žádný, mě známý nástroj s grafickým jazykem pro roboty neumožňuje. Částečně lze dosáhnout opačného směru a zkompilovanou službu znovu po-



Obrázek 12: MRDS VPL – program pro detekci překážky

užít v dalším VPL diagramu. Ale pouze bez možnosti zobrazení vnitřní struktury bloku. Další obrovskou výhodou je možnost rekurze, uvnitř služby můžeme volat znovu tuto službu. Tím se zjednoduší návrh služby.

4.4 Visual Simulation Environment (VSE)

Tento nástroj umožňuje simulovat služby vytvořené v MRDS bez nutnosti vlastnit jakýkoliv robotický hardware. Simulace ve 3D podporuje základní fyzikální vlastnosti. K dispozici jsou různé 3D prostředí a 3D modely robotů. Problémem mohou být speciální konstrukce robotů, kdy je nutno vytvořit si vlastní model nebo sehnat již vytvořený někde na internetu. Ve složce $MRDS\backslash samples\backslash SimulationTutorials\backslash$ nalezneme tutoriály jak vytvořit simulaci. Některé připravené simulace lze spustit přímo z nabídky *Start* → *Všechny programy* → *MRDS* → *Visual Simulation Environment 2008 R2*. V této práci se nezabývám simulacemi vytvořených služeb, proto nebudu podrobně popisovat tento nástroj. Ukázka nástroje se simulací služby s robotem NXT, viz Obrázek 29 na straně i v příloze.

5 Jednoduché úlohy ve VPL

Součástí této práce jsou ukázky vytváření jednoduchých programů ve VPL. Jako vzor byly vybrány některé ukázkové programy napsané v jazyce RoborC, popsaném v kapitole 3. Můžeme tak porovnat rozdíl složitosti mezi grafickým a strukturovaným programovacím jazykem. Obecně lze považovat grafické programování pro začátečníky za snadnější. Někdy ale složitější věci se snadněji řeší pomocí strukturovaného programování. Je nutné ovšem strukturovaný jazyk dobře znát. Ačkoliv RoborC dodává spoustu vzorových úloh, některé byly příliš komplikované na to, aby šly vytvořit ve VPL.

5.1 Úlohy s využitím motorů

Tyto úlohy sice popisují ukázku využití motorů, primárně ale slouží jako ukázka použití cyklů a základních konstrukcí programu.

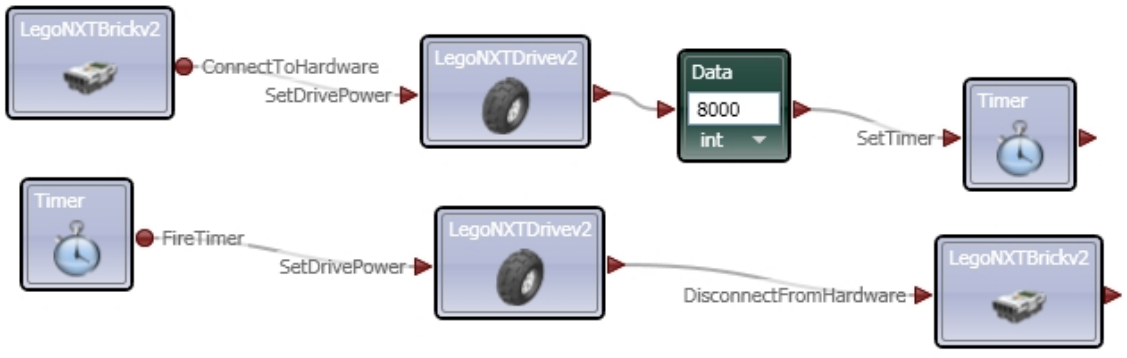
5.1.1 Program „nxt_moving_forward“

Asi nejjednodušší program, který jen spustí oba motory maximální rychlostí vpřed a po 8s se zastaví. Zdrojový kód programu viz Výpis 3.

```
task main(){
  motor[motorA] = 100;
  motor[motorB] = 100;
  wait1Msec(8000);
}
```

Výpis 3: Program „nxt_moving_forward“ v RobotC

Pomocí VPL vytvoříme podobný program konající stejnou činnost. Po připojení k NXT nastavíme rychlost motorů na maximum a nastavíme časovač *Timer* na 8s. Po tuto dobu pojede robot rovně, poté se zastaví a program se ukončí odpojením od NXT, viz Obrázek 13. Nastavení rychlostí není na obrázku vidět, jde vidět až při kliknutí na spojovací linii mezi bloky. Rychlost ve VPL je nutno dělit 100.



Obrázek 13: Program „nxt_moving_forward“ ve VPL

5.1.2 Program „nxt_modifiers“

Program ukazuje, jak nastavit motory na různé hodnoty s různými intervaly čekání mezi přenastavením rychlosti. Nejprve nastaví motory na maximální rychlost vpřed a provádí tento úkon 2s, poté oba motory jedou rychlostí 65 ze 100 vzad po dobu 4s. Nakonec se provede rotace pomocí nastavení jednoho motoru vpřed a druhého vzad po dobu 2,5s. Program napsaný v RoborC, viz Výpis 4.

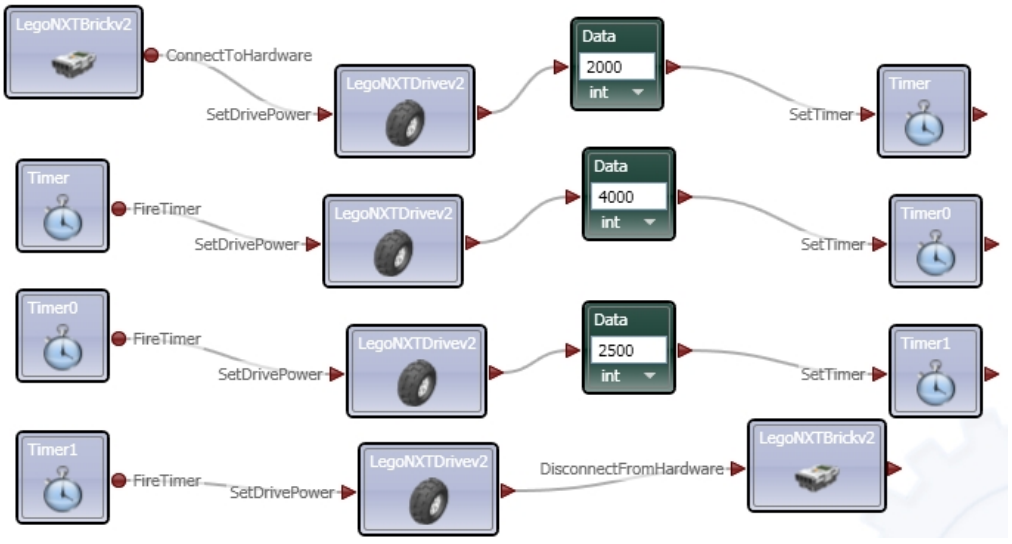
```
task main(){
  motor[motorA] = 100;
  motor[motorB] = 100;
  wait1Msec(2000);

  motor[motorA] = -65;
  motor[motorB] = -65;
  wait1Msec(4000);

  motor[motorA] = 75;
  motor[motorB] = -75;
  wait1Msec(2500);
}
```

Výpis 4: Program „nxt_modifiers“ v RobotC

Ve VPL je potřeba použít tři časovače. Program je celkem jednoduchý, viz Obrázek 14, po připojení robota dojde k nastavení maximální rychlosti vpřed a nastavení časovače *Timer* na 2s. Po uplynutí doby se spustí rutina prováděná časovačem, která nastaví rychlost motorů na zpětný chod a nastaví časovač *Timer0* na 4s. Pak se obdobně nastaví rychlost a třetí časovač, po kterém dojde k ukončení programu.



Obrázek 14: Program „nxt_modifiers“ ve VPL

5.1.3 Program „nxt_loop_1_while“

Tento program je jednoduchou ukázkou používání cyklu *while*. Ve VPL ovšem cykly nemáme, proto je třeba nahradit je jinou konstrukcí. Program napsaný v RoborC viz Výpis 5. Program v cyklu 5krát spustí oba motory vpřed po dobu 2 sekund, následně spustí jeden motor vpřed a druhý vzad a vykonává rotaci 750ms.

```
task main(){
  int count = 0;

  while(count < 5){
    motor[motorA] = 75;
    motor[motorB] = 75;
    wait1Msec(2000);

    motor[motorA] = 75;
    motor[motorB] = -75;
    wait1Msec(750);

    count++;
  }
}
```

Výpis 5: Program „nxt_loop_1_while“ v RobotC

Ve VPL je situace o něco komplikovanější, cyklus *while* musíme nahradit blokem *Merge* a proměnné inicializovat pomocí bloku *Variable*. Zároveň nelze čekání jednoduše realizovat pomocí *wait*, ale je nutno přidat blok *Timer*. Celý program přepsaný do VPL viz Obrázek 30 na straně ii v příloze.

Na začátku dojde k připojení zařízení a nastavení rychlostí obou motorů vpřed. Mezi tuto akci je nutno přidat blok *Merge* jako náhradu cyklu *while*. Proměnnou *count* pro cyklus je potřeba inicializovat jako samostatný blok. Po nastavení rychlosti se nastaví první časovač *Timer* na 2s. Až tato doba uplyne, nastaví se rychlost jednoho motoru vpřed a druhého motoru vzad. Dojde k nastavení druhého časovače *Timer0* na 750ms. Po uplynutí této doby dojde k navýšení hodnoty *count* o jedna. Následně se testuje hodnota *count*, náhrada za testování v cyklu *while*. Pokud je hodnota menší než pět, řízení se předává do bloku *Merge*, jinak dojde k ukončení programu. Všimněte si, jak se program zkomplikoval. Např. jednořádkové přičtení hodnoty *count++* je zde nahrazeno třemi bloky.

5.2 Úlohy se světelným senzorem

Zde uvedeme ukázky několika aplikací použití světelného senzoru.

5.2.1 Program „nxt_line_track“

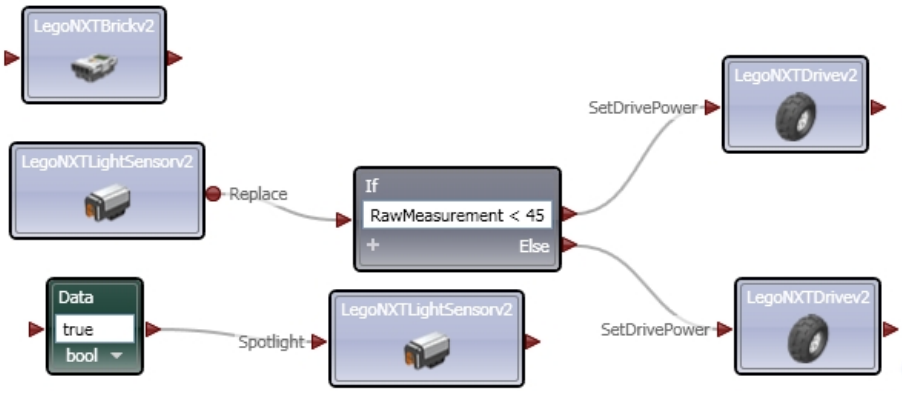
Jedná se o jednoduchý program pro sledování černé čáry pomocí světelného senzoru popsaného v kapitole 2.2.2. Pokud robot snímá světelným senzorem černou barvu, spustí jeden motor a druhý ponechá vypnutý. V opačném případě se prohodí nastavení motorů

a tím robot tzv. pohybem *„ZIG – ZAG“* sleduje čáru. Program je v nekonečné smyčce *while*, tím je zaručeno neustálé sledovaní čáry. Program napsaný v RoborC viz Výpis 6.

```
task main(){
  while(true){
    if(SensorValue( lightSensor ) < 45){
      motor[motorA] = 75;
      motor[motorB] = 0;
    }
    else{
      motor[motorA] = 0;
      motor[motorB] = 75;
    }
  }
}
```

Výpis 6: Program „nxt_line_track“ v RobotC

Ve VPL nemusíme cyklus *while* nahrazovat blokem *Merge*, protože světelný senzor má nastavenu *pooling* frekvenci, která určuje interval nového snímání hodnoty. Blok *If – Else* se nahradí blokem *If*. Je nutné také zapnout senzor do aktivního režimu nastavením hodnoty *Spotlight* na *true*. Bohužel ve VPL nedosáhneme stejné rychlosti jako u programu v RoborC. Díky zahlcení fronty zpráv NXT přes bluetooth nám robot při stejné rychlosti vyjede mimo, tento problém je vysvětlen v kapitole 6.1. Nezbude nic jiného, než nastavit rychlost nižší, viz Obrázek 15.



Obrázek 15: Program „nxt_line_track“ ve VPL

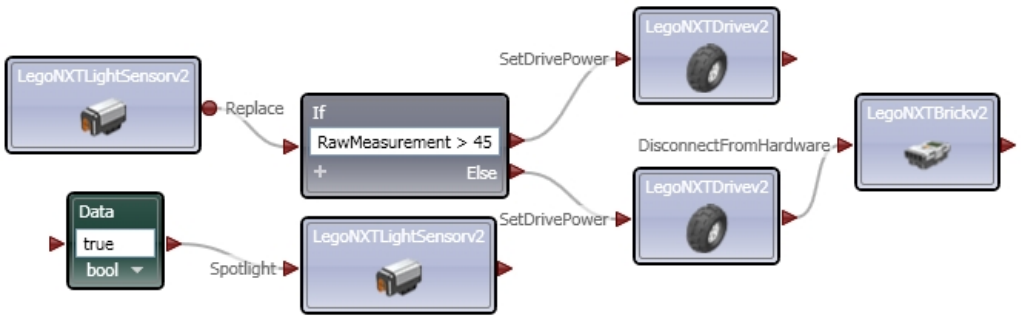
5.2.2 Program „nxt_wait_for_dark“

Další jednoduchý program s využitím světelného senzoru: zde robot jede neustále rovně, dokud nezaznamená senzorem černou barvu. Program napsaný v RoborC viz Výpis 7. Oproti předešlému programu je *if* nahrazen cyklem *while*.

```
task main(){
  while((SensorValue( lightSensor ) < 45){
        motor[motorA] = 100;
        motor[motorB] = 100;
  }
  motor[motorA] = 0;
  motor[motorB] = 0;
}
```

Výpis 7: Program „nxt_wait_for_dark“ v RobotC

Zde, ačkoliv program v RoborC dělal něco naprosto odlišného, jde ve VPL o prakticky stejnou věc. Dokud zaznamená bílou jede vpřed, pokud zaznamená černou zastaví a ukončí program, viz Obrázek 16. Rychlosti bohužel nejsou vidět, proto vypadá obrázek dosti podobně jako předešlý.



Obrázek 16: Program „nxt_wait_for_dark“ ve VPL

5.2.3 Program „nxt_line_track_timer“

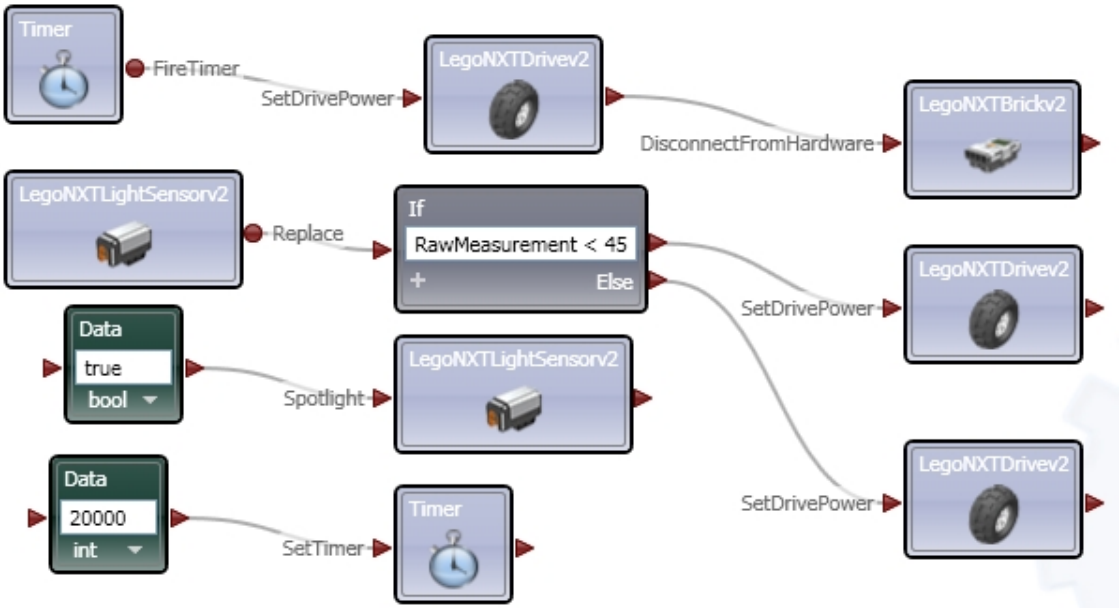
Obdobný program na sledování čáry, ale s časovým omezením: v RoborC je použita direktiva *time1(T1)*, což je přímý přístup k prvnímu časovači mikroprocesoru. Po dokončení určité doby je vyvoláno hardwarové přerušení. Program napsaný v RoborC viz Výpis 8.

```
task main(){
  time1(T1) = 0;

  while(time1(T1) < 20000){
    if (SensorValue( lightSensor ) < 45){
      motor[motorA] = 75;
      motor[motorB] = 0;
    }else{
      motor[motorA] = 0;
      motor[motorB] = 75;
    }
  }
}
```

Výpis 8: Program „nxt_line_track_timer“ v RobotC

Ve VPL nahradíme časovač blokem časovače. Jinak je program podobný obyčejnému programu na sledování čáry, viz Obrázek 17.



Obrázek 17: Program „nxt_line_track_timer“ ve VPL

5.3 Úlohy s ultrazvukovým senzorem

Zde si ukážeme použití ultrazvukového senzoru popsaného v kapitole 2.2.5.

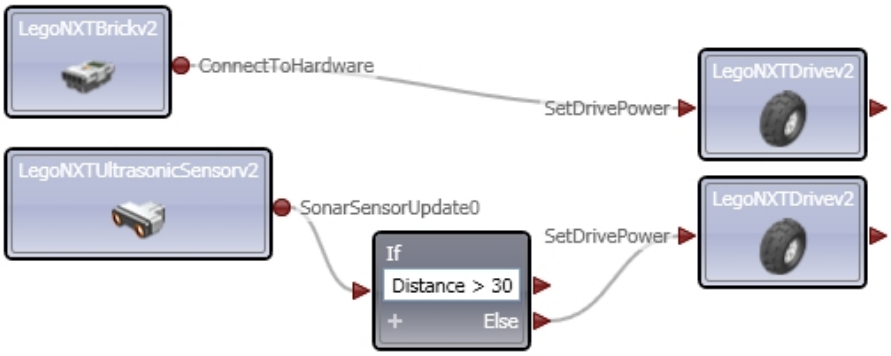
5.3.1 Program „nxt_obstacle_detect“

Detekce překážky je v podstatě obdobná, jako detekce černé čáry. Místo cyklu *while* je použit cyklus *do – while*. Program napsaný v RoborC viz Výpis 9.

```
task main(){
  do{
    motor[motorA] = 75;
    motor[motorB] = 75;
  }while(SensorValue(sonarSensor) > 30);
}
```

Výpis 9: Program „nxt_obstacle_detect“ v RobotC

Ve VPL je opět detekce překážky podobná jako detekce čáry. Místo využití druhé větve bloku *If* je použit blok *LegoNXTBrickV2* pro spuštění obou motorů vpřed. Program ve VPL viz Obrázek 18 na následující straně.



Obrázek 18: Program „nxt_obstacle_detect“ ve VPL

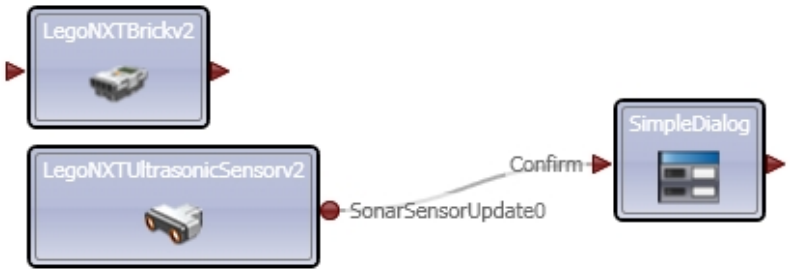
5.3.2 Program „nxt_sonar_display“

Toto je jednoduchý program pro zobrazení vzdálenosti objektu od ultrazvukového senzoru. V nekonečném cyklu se načítá hodnota ze senzoru, zapisuje se na diplej NXT. Po 1s se displej smaže a akce se opakuje. Program napsaný v RoborC viz Výpis 10.

```
task main(){
  int x=0;
  while(true){
    x = SensorValue(sonarSensor);
    nxtDisplayTextLine(1,"Sonar␣Reading:%d",x);
    wait1Msec(1000);
    eraseDisplay();
  }
}
```

Výpis 10: Program „nxt_sonar_display“ v RobotC

RoborC umožňuje přímý zápis na displej. Ve VPL bohužel pro zápis textu na displej NXT neexistuje žádná služba. Můžeme využít zobrazení hodnoty přímo na monitor pomocí bloku *SimpleDialog*. Existuje sice možnost spuštění programu NXT–G, který přijme bluetooth zprávu a zapíše její obsah na displej. To ale není čisté řešení pomocí VPL. Program ve VPL viz Obrázek 19.



Obrázek 19: Program „nxt_sonar_display“ ve VPL

5.4 Úlohy s rotacemi

Nyní si ukážeme aplikace, které pro určení úhlu otočení využívají senzorů rotace motorů. Každý motor má svůj senzor rotace, zaznamenává rotaci s přesností na 1°.

5.4.1 Program „nxt_point_turns_enc“

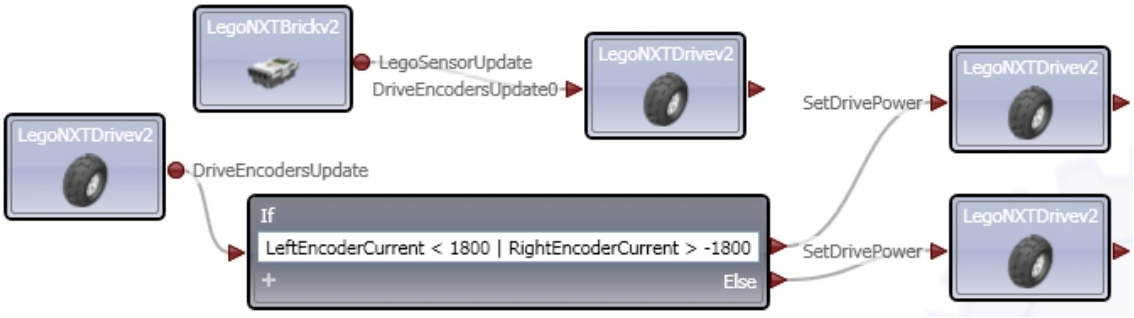
V RoborC pomocí direktivy *nMotorEncoder* přistupujeme k hodnotám čítače otáček motoru. Nejprve je ale nutno čítač vynulovat. V jednoduchém programu vynulujeme čítače obou motorů a necháme robota točit se kolem své osy, dokud jeden z motorů nedosáhne otáčením 1800°. Program napsaný v RoborC viz Výpis 11.

```
task main(){
    nMotorEncoder[motorA] = 0;
    nMotorEncoder[motorB] = 0;

    while(nMotorEncoder[motorA] < 1800 || nMotorEncoder[motorB] > -1800){
        motor[motorA] = 100;
        motor[motorB] = -100;
    }
}
```

Výpis 11: Program „nxt_point_turns_enc“ v RobotC

Ve VPL musíme také vynulovat hodnotu čítačů, to provedeme pomocí metody *DriveEncodersUpdate* bloku *LegoNXTDriveV2*. Poté můžeme v bloku *If* přistupovat přímo k hodnotám čítačů pomocí proměnných *RightEncoderCurrent* a *LeftEncoderCurrent*. Program ve VPL viz Obrázek 20.



Obrázek 20: Program „nxt_point_turns_enc“ ve VPL

5.4.2 Program „nxt_line_track_rotations“

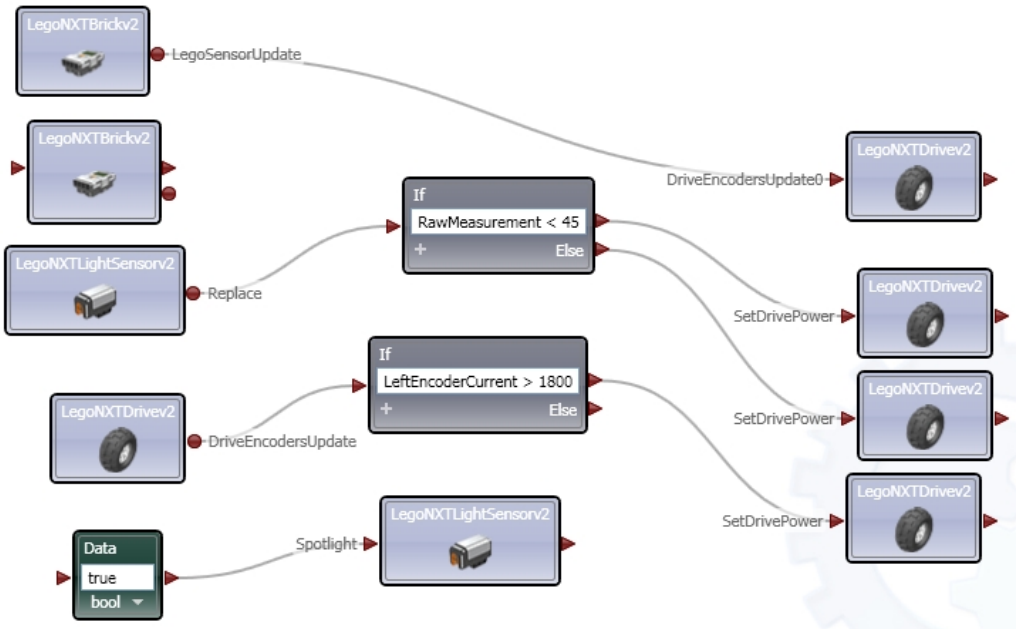
Tento program ukazuje, jak při sledování čáry omezit dráhu sledování počtem rotací motoru. Sledování čáry probíhá dokud se motor *A* neotočí o 1800°. Program napsaný v RoborC viz Výpis 12.

```
task main(){
  nMotorEncoder[motorA] = 0;

  while(nMotorEncoder[motorA] < 1800){
    if (SensorValue( lightSensor ) < 45){
      motor[motorA] = 75;
      motor[motorB] = 0;
    }else{
      motor[motorA] = 0;
      motor[motorB] = 75;
    }
  }
}
```

Výpis 12: Program „nxt_line_track_rotations“ v RobotC

Zde je podobná kombinace příkladu sledování čáry a předchozího příkladu omezení rotací. Musí se vynulovat čítač otáček jako v předcházejícím případě. Program ve VPL viz Obrázek 21.



Obrázek 21: Program „nxt_line_track_rotations“ ve VPL

6 Aplikace NXT Labyrinth Solver

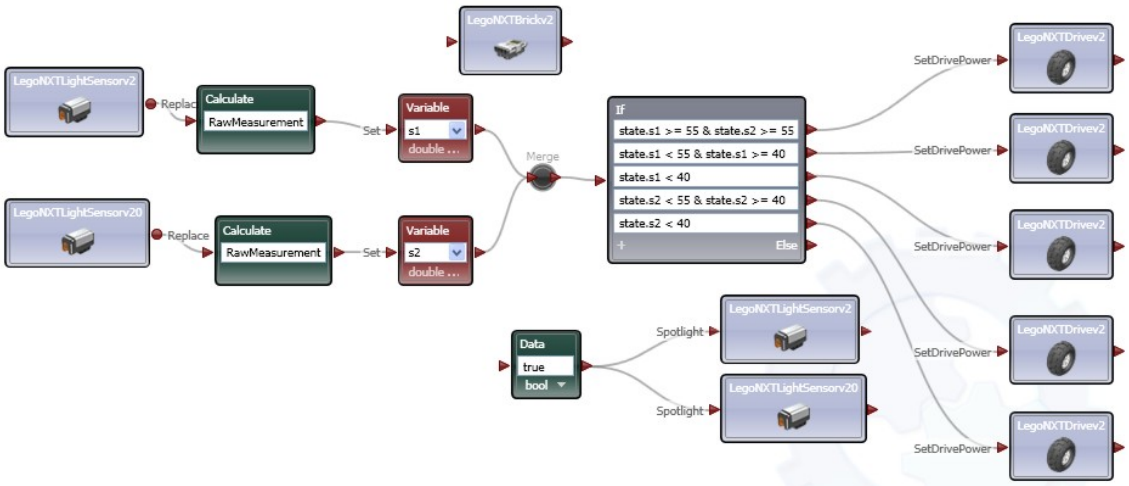
Častou aplikací v robotickém světě je implementace řešení bludiště. Typickým bludištěm bývá bludiště tvořené stěnami nebo bludiště složené z křižovatek. Robot se pomocí svých senzorů musí orientovat v bludišti a najít sám bez zásahu člověka cestu k východu. My jsme se rozhodli implementovat bludiště tvořené křižovatkami s robotem LEGO NXT. Implementace bludiště je řešena využitím dvou prostředí MRDS a NXT–G.

6.1 Návrh řešení bludiště

Bylo potřeba navrhnout konstrukci robota a návrh implementace pro řešení takovéhoto bludiště. K dispozici jsme měli 2 světelné senzory a samozřejmě robota LEGO NXT. Pro implementaci jsme zvolili nástroj MRDS VPL popsaný v kapitole 4.3. Nejprve bylo potřeba implementovat sledování čáry. Pro sledování čáry sice stačí jen jeden senzor, ale robot pak dělá tzv. *„ZIG – ZAG“* pohyb. Pokud senzor snímá černou zatáčí doprava, pokud snímá bílou zatáčí doleva. Tak vykonává kyvadlový pohyb, kterým se posouvá vpřed. Toto se dá řešit různými algoritmy, které omezují tento jev. Elegantní je ale využít dva světelné senzory, jak je popsáno v [6]. Tam popisují řešení pomocí následujících pěti podmínek:

1. Pokud oba senzory snímají bílou (např. $> 75\%$), jeď rovně.
2. Pokud levý senzor snímá částečně černou (např. 25-75%), jeď mírně doleva.
3. Pokud levý senzor snímá úplně černou (např. $< 25\%$), jeď prudce doleva.
4. Pokud pravý senzor snímá částečně černou, jeď mírně doprava.
5. Pokud pravý senzor snímá úplně černou, jeď prudce doprava.

Po přepsaní algoritmu do VPL vypadal diagram zhruba následovně, viz Obrázek 22.



Obrázek 22: MRDS VPL – Sledování čáry s dvěma senzory

Bohužel při spuštění programu vypadala jízda robota dost trhaně, navíc častokrát robot přejel čáru a jel mimo dráhu. Zkoušeli jsme zvyšovat pooling frekvenci (rychlost snímání) snímače a snižovat rychlosti motorů, ale nebylo to dokonalé řešení. Robot jel pak příliš pomalu a někdy stejně nezaregistroval čáru, a proto vyjel mimo. Zkoušeli jsme obdobný program napsat v NXT–G a v RobotC, tam byla rychlost o mnoho vyšší a bez chyb. Robot nikdy nevyjel z dráhy. Po dlouhém hledání chyby jsme narazili na článek s analýzou bluetooth komunikačního protokolu LEGO NXT [12], který napsal Sivan Toledo. V něm se píše o tom, že NXT má frontu jen pro pět zpráv a pokud se zahltí, dojde k zahazování nejstarších zpráv. Z toho jsme usoudili, že právě toto je příčinou celého problému. Možná že u robotů jiných platforem k tomuto problému nedochází.

Celé to vzniklo tím, že robot musel neustále číst data ze světelného senzoru a posílat je přes bluetooth do počítače. Zároveň musel neustále přijímat zprávy z počítače s korekcemi rychlostí motorů. Tím se stane aplikace téměř neproveditelná v reálném čase. Uvažovali jsme nad tím, jakým způsobem snížit četnost zasílání zpráv mezi počítačem a robotem. Napadlo nás, že by možná stačilo, pokud by informace ze snímačů nemusel odesílat. Jen by zasílal informace o stavu, ve kterém se nachází. Tím by byl problém vyřešen, ale pořád jsme nepřicházeli na způsob, jak to za pomocí VPL vyřešit.

Posléze jsme našli mezi VPL tutoriály ve složce *MRDS\samples\VplExamples\LEGO\MsgReadWrite* příklad, který ve VPL text vyplněný v textovém poli na počítači zapisuje na displej robota. Po podrobném prozkoumání příkladu jsme zjistili, jakým způsobem funguje. Byl rozdělen na dvě části. Jedna byla nahrána v počítači, druhá pomocí NXT–G v paměti kostky. Program fungoval následovně. Na začátku byl odeslán přes bluetooth příkaz na spuštění programu v kostce. Program v kostce obsahoval smyčku, kde čekal na příchozí zprávu z bluetooth, tu zobrazil na displej. Následovalo ve VPL zobrazení dialogu s textovým polem na počítači, tam uživatel zadal text a stiskl tlačítko *OK*. Počítač odeslal zprávu přes bluetooth. Celý cyklus se pak opakoval. Napadlo nás podobný princip využít pro omezení zasílání zpráv mezi počítačem a robotem.

6.1.1 Návrh konstrukce robota

Měli jsme robota v konstrukci *TriBot*, viz Obrázek 23 na následující straně. Ten měl pouze jen jeden světelný senzor, navíc měl chapadla na uchopení míčku, tlačítkový senzor pro zjištění, že došlo k uchopení míčku, mikrofon a ultrazvukový senzor pro měření vzdálenosti. Jeden světelný senzor byl uprostřed, my jsme se ale rozhodli použít senzory dva.

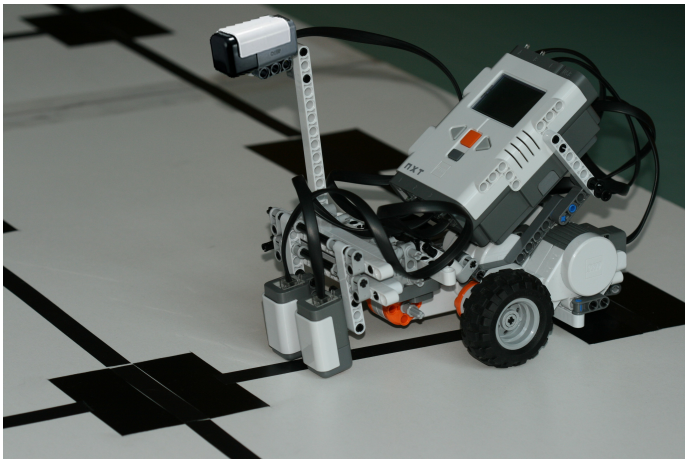
Bohužel do konstrukce *TriBota* se nedal nijak připevnit druhý světelný senzor. Odstranili jsme proto celou přední část konstrukce *TriBota* – chapadlo, ultrazvukový senzor, tlačítkový senzor a světelný senzor. Navíc jsme nepotřebovali mikrofon, tak jsme jej taky odstranili. Bylo potřeba navrhnout vhodné uchycení pro dva světelné senzory. První pokus byl sice úspěšný, ale výsledná konstrukce nedovolovala změnu výšky a rozteče mezi senzory. Kvůli možnosti testování různých sestav pro různé šířky pásek a druhy pásek použitých na cestu pro bludiště, bylo potřeba vytvořit škálovatelnou konstrukci uchopení senzorů.

Nová konstrukce si vyžádala několik hodin zkoumání použitelných LEGO součástek. Měli jsme k dispozici součástky z kompletního LEGO NXT 1.0 kitu. Výsledná konstrukce

Obrázek 23: LEGO NXT – konstrukce TriBot [1]

se dala jednoduše posouvat a po dlouhém testování jsme nechali světelné senzory ve výšce cca 4mm nad zemí a rozteč mezi světelnými senzory pro pásku šíře 15mm byla ideální přibližně 350 – 400mm (bráno střed levého senzoru ⇔ střed pravého senzoru). Existuje mnoho faktorů a nespočet možností nastavení rychlostí motorů při určité intenzitě snímané hodnoty světelného senzoru a rozteče mezi senzory. Tato část testování zabrala spoustu času a nedokážeme říct, zda je tou nejlepší variantou.

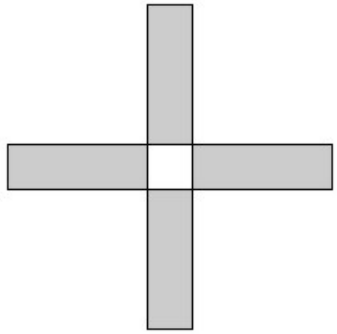
Katedra získala nový HiTechnic kompas senzor popsaný v 2.2.8 a rozhodli jsme se ho takto uplatnit v řešení bludiště, byt' to nebylo nezbytně nutné. Bylo nutné ho umístit 15cm od motorů a 10cm od kostky. Pomocí dlouhého bílého dílu LEGA jsme ho umístili přímo nad světelné senzory. Obrázek 24 znázorňuje celou konstrukci robota.



Obrázek 24: LEGO NXT – konstrukce se dvěma světelnými senzory a kompasem

6.1.2 Návrh konstrukce bludiště

Návrh bludiště byl závislý na typu bludiště (v našem případě bludiště tvořené křižovatkami), použitých senzorech a konstrukci robota. Díky dvěma světelným senzorům jsme sice měli robota, který je schopen rychle sledovat čáru, ale standardní křižovatku pro jeden světelný senzor, viz Obrázek 25, (šedá barva je černá páska) není schopen řešit.



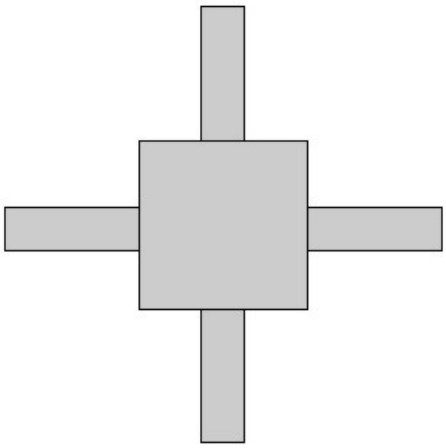
Obrázek 25: Křižovatka pro jeden světelný senzor

Robot v bludišti s křižovatkami pro jeden světelný senzor sleduje čáru a jakmile uvidí velmi nízký odstín šedi (bílý čtverec v křižovatce), zjistí že se jedná o křižovatku. Posune se dopředu a začne prozkoumávat křižovatku, do kterých směrů vedou cesty. Robot se dvěma světelnými senzory by zareagoval chybně. Bílý čtverec uprostřed by neviděl, křižovatku by nezaregistroval a možná by jen projel rovně nebo zatočil do nějakého směru. Pro dva světelné senzory musela být navržena nová křižovatka. Napadlo nás, že robot snímá dvěma senzory vždy jednu ze tří kombinací, přičemž čtvrtá kombinace nebyla využita:

1. Bílá levý senzor – bílá pravý senzor (jede dopředu rychle, pásku má přímo mezi senzory).
2. Černá levý senzor – bílá pravý senzor (jede doleva, pásku má pod levým senzorem).
3. Bílá levý senzor – černá pravý senzor (jede doprava, pásku má pod pravým senzorem).
4. Černá levý a pravý senzor – nevyužitá kombinace.

Napadlo nás tuto kombinaci využít pro rozpoznání křižovatky. Vytvořili jsme křižovatku z černé pásky ve tvaru čtverce, viz Obrázek 26 na následující straně, (šedá barva je černá páska). Velikost jsme zvolili tak, aby oba senzory zaznamenaly černou barvu. Velikost musela být dobře zvolena s ohledem na konstrukci robota. Kdyby byla křižovatka příliš úzká, robot by ji nezaznamenal a jen by zatočil. Naopak příliš široká křižovatka by způsobila, že při průzkumu cest by robot zaznamenal i rohy křižovatky. To by vedlo k chybnému určení směru sousední křižovatky. Vzdálenost mezi levým vnějším krajem levého senzoru a pravým vnějším krajem pravého senzoru činila 65mm. Rozhodli jsme se pro rozměr křižovatky $90 \times 90mm$, robot správně registroval křižovatku a při otáčení kolem

své osy zaznamenával pouze cesty vedoucí z křižovatky. Vzdálenost mezi křižovatkami byla stanovena na minimálně 150mm, aby se robot stihl vyrovnat a nezačal průzkum křižovatky šikmo. Tak by docházelo k chybám při určování směru cest vycházejících z křižovatky. Pro cesty byla použita páska šíře 15mm.



Obrázek 26: Navržená křižovatka pro dva světelné senzory

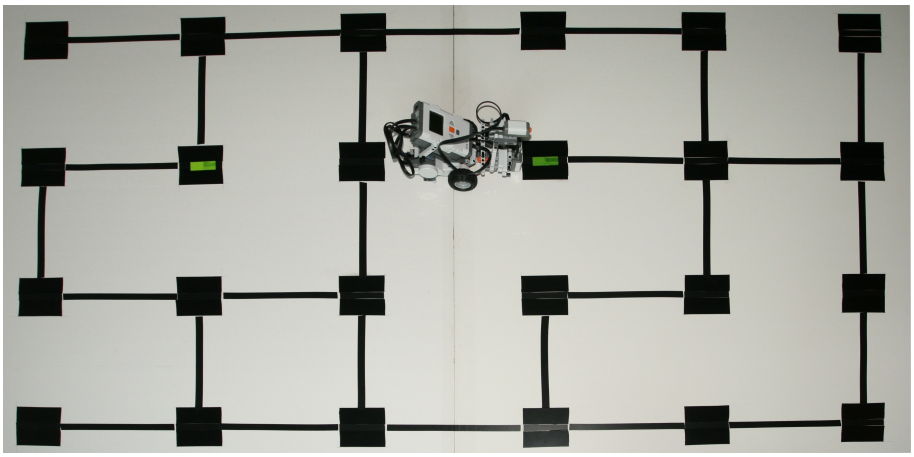
Křižovatka může kromě cesty zpět obsahovat žádnou, jednu, dvě nebo tři cesty do dalších křižovatek. Slepá cesta můsí být zakončena křižovatkou. Vzdálenost mezi křižovatkami musí být vždy jednotková (tím je myšleno stejná vzdálenost mezi křižovatkami), protože robot nepočítá ujetou vzdálenost a nemohl by pak zjistit svoji aktuální souřadnici v matici bludiště. Maticí bludiště rozumíme dvourozměrné pole, ve kterém je na souřadnici *[x,y]* křižovatka. Posun robota o jednu jednotku (na další křižovatku) bude znamenat posun o jednu souřadnici v matici bludiště (např. z *[0,0]* na *[1,0]*). Pokud chceme například udělat cestu vzdálenosti dvou jednotek, vytvoříme mezi dvěma křižovatkami další křižovatku. Ta bude mít cestu zpět k první křižovatce a cestu rovně k druhé křižovatce.

Dalším problémem byl typ použité pásky, obyčejná páska se příliš leskla a občas ji robot nezaznamenal, pokud se okolní světlo od ní odráželo. Nakonec jsme použili elektroizolační pásku, která je více matná a poměrně dobře se s ní pracuje. K chybám při snímání povrchu již nedocházelo.

Rozměr matice bludiště může být libovolný, např. i 2×2 křižovatky, čím větší rozměr, tím delší doba prozkoumávání bludiště. My jsme vytvořili bludiště o rozměrech matice 4×6 křižovatek, viz Obrázek 27 na následující straně. Pozn. startovní křižovatka a cílová křižovatka se nemusí od ostatních nijak lišit, protože se zadávají do programu jejich pozice v matici bludiště. Robot si pak počítá svou pozici v matici bludiště. Na obrázku je startovní a cílová křižovatka označena jen pro lidské oko.

6.1.3 Návrh implementace aplikace

Z příkladu VPL tutoriálu jsme zjistili, jakým způsobem funguje spouštění programů uložených v LEGO NXT přes VPL. Napadlo nás rozdělit aplikaci na dvě základní části, první



Obrázek 27: Navržené bludiště

bude uložena v robotovi (napsána v NXT–G viz kapitola 2.3), bude obstarávat pohyb robota v bludišti a prozkoumávání křižovatek.

První část nebude vědět nic o tom, jak veliké je bludiště, jakým směrem se vydat, na jaké pozici se robot právě nachází. Bude to jen jakási podpora pro běh. Bude jen zasílat přes bluetooth informace o aktuálním stavu. Po úvaze nad tím, co vše robot v bludišti potřebuje, bylo sepsáno několik základních programů nutných[1] pro tuto aplikaci. Seznam je následující:

- sledování čáry – pro pohyb mezi křižovatkami,
- průzkum křižovatky – prozkoumání tvaru křižovatky (které cesty z ní vedou),
- průjezd již prozkoumanou křižovatkou,
- otočení vpřed,
- otočení doprava,
- otočení doleva,
- otočení zpět.

Druhá část bude v počítači (napsaná ve VPL/C#), bude se starat o logiku celé aplikace, a tím bude tvořit „mozek“ celé aplikace. Bude si pamatovat pozici robota v matici bludiště, pomocí spouštění programů ovládat směr pohybu robota a pamatovat si, které křižovatky neprozkoumal. To bude provádět na základě vyhodnocení bluetooth zpráv, které od robota přijdou. Po prozkoumání bludiště vypočte díky znalosti kompletní topologie bludiště nejkratší cestu z počáteční pozice do cílové pozice.

Chování robota popisuje diagram viz Obrázek 31 na straně iii v příloze. Celý program bude fungovat následovně. Nejprve se spustí program na sledování čáry. Až robot narazí

[1] Později přibyly další, ale ty přímo nesouvisí s řešením bludiště např. kalibrace aj.

na křižovatku, odešle přes bluetooth zprávu, že je na křižovatce. Hlavní program přijme zprávu, podívá se do své paměti a zjistí, zda je křižovatka prozkoumaná. Pokud ano, zašle přes bluetooth příkaz na spuštění programu na průjezd křižovatkou, pokud ne zašle příkaz na spuštění programu na průzkum křižovatky. Program na průjezd křižovatkou nechá robota projet křižovatku a odešle zprávu, že projel křižovatku. Program na průzkum křižovatky otočí robota jedenkrát kolem své osy a zašle zprávy o směrech, které z křižovatky vychází. Nakonec odešle zprávu o dokončení průzkumu. Hlavní program si informace o směrech vycházejících z křižovatky uloží do paměti a rozhodne směr průzkumu. Pořadí prozkoumávání je rovně, doleva, doprava, pokud jsou všechny směry prozkoumány vrátí se zpět. Procházení probíhá algoritmem prohledávání do hloubky, dokud robot může prohledávat dál, nevrací se. Po otočení do zvoleného směru robot vyšle zprávu, že je otočen. Hlavní program přijme zprávu a zašle příkaz na spuštění programu sledování čáry. Celý cyklus se opakuje až do nalezení cíle, tam, pokud je nastaveno hledání nejkratší cesty, pokračuje v prohledávání. Pro nalezení nejkratší cesty musí robot prohledat celé bludiště a vrátit se zpět na startovní pozici. Pak vypočte nejkratší cestu pomocí algoritmu prohledávání do šířky a vydá se po ní. Díky jednotkové vzdálenosti je prohledání snadnější (cesty do sousedních křižovatek mají stejnou délku).

6.2 Implementace řešení bludiště

Implementace této aplikace byla trochu nestandardní implementací. Už proto, že bylo nutno vytvořit programy ve více programovacích jazycích. O to horší bylo, že některé jazyky byly grafické a použití softwaru pro správu verzí, např. SVN (Subversion) bylo sice možné, ale zdrojové soubory u některých prostředí se ukládají na více míst a navíc se jednotlivé verze nedají porovnávat. Až při práci v C# byla použita SVN.

6.2.1 Programy v NXT–G

Nejprve jsme začali psát programy v NXT–G, ty měly jednu obrovskou výhodu, daly se snadno a rychle otestovat na robotovi a spouštět bez nutnosti mít kompletní řešení. Protože jsme si řešení rozdělili do jednotlivých podprogramů, stačilo je vytvářet postupně. Vznikly následující programy:

- linefollow2s.rxe – sledování čáry,
- explore_cross.rxe – průzkum křižovatky,
- run_over_cross.rxe – průjezd již prozkoumanou křižovatkou,
- center_straight.rxe – otočení vpřed,
- center_right.rxe – otočení doprava,
- center_left.rxe – otočení doleva,
- center_back.rxe – otočení zpět.

Nejprve bylo potřeba přizpůsobit algoritmus v programu sledování čáry *linefollow2s* pro najetí na křižovatku. V NXT–G to znamenalo přidat novou podmínku pro možnost, kdy oba senzory zaznamenají černou barvu. Uvnitř této podmínky bylo potřeba zastavit motory a vyslat zprávu přes bluetooth s textem *„cross“*, aby program v počítači věděl, že robot dorazil na křižovatku a spustil podle toho další program.

Asi nejdůležitějším programem pro řešení bludiště je program *explore_cross* – průzkum křižovatky. Zde by jakýkoliv nedostatek vedl k fatální chybě a nemožnosti pokračovat v prohledávání bludiště se správným výsledkem. Jako příklad můžeme uvést nezaregistrování cesty vedoucí z křižovatky, která může být jedinou vedoucí k cíli. Robot nejprve musí vjet přímo na křižovatku, pak může začít z průzkumem. Prvně jsme používali k rozpoznání směru cest HiTechnic kompas senzor popsaný v kapitole 2.2.8. Vždy jsme uložili do paměti úhel na počátku rotace a vzhledem k němu vypočítávali směry okolních cest. Bohužel občas docházelo k nepřesnostem a nedokázali jsme nastavit správnou hranici mezi dvěma sousedními směry. Jednou robot zjistil směr správně, podruhé špatně. Asi to bylo způsobeno rušením motorů, možná by pomohlo vzdálenější umístění kompasu nebo kalibrace. Rozhodli jsme se vyřešit tento problém pomocí senzoru rotace motoru. Nevýhodou je, že pomocí senzor rotace motoru získáme pouze stupně otočení hřídele kola, kdežto pomocí kompas senzoru jsme získali přímo stupně otočení robota. Zjistili jsme si úhel, který senzor zaznamená při rotaci robota kolem své osy. Ten jsme rozdělili na čtyři stejné kvadranty a podle toho určovali směry. K chybám již nedocházelo. Pokaždé, když robot zjistil směr cesty z křižovatky, zaslal jednu zprávu podle směru *„right“, „left“ a „straight“* (o existenci zpátečního směru není potřeba informovat). Na konci průzkumu robot odeslal bluetooth zprávu *„explored“*.

Po dokončení průzkumu jsme potřebovali vytvořit programy, pomocí kterých by hlavní program mohl robotu dát pokyn ke změně směru. Tyto programy *center_right, center_left, center_straight, center_back* otočily robota do požadovaného směru. Zde jsme sice použili kompas senzor. Pokud robot neměl čáru přímo mezi senzory, tak došlo k otočení do špatného směru. Kompas to zjistil a nechal robota otočit do správné pozice. Počítali jsme s tím, že křižovatka má opravdu tvar takový, jak ho robot prozkoumal. Například kdyby se měl robot otočit doleva a cesta by tam nebyla, otočil by se zpět. Samozřejmě by to vedlo k chybnému prohledávání, ale to nezaznamenaná cesta taky. Robot se proto při otáčení doleva, doprava a zpět jen otočí, a dál nekontroluje správnost úhlu otočení. Nakonec dojde k vyrovnání senzorů přesně na střed čáry, využíváme absolutní hodnotu z rozdílu hodnot zjištěných oběma senzory. Pokud je blízká nule, centrování je ukončeno a robot zašle zprávu *„centred“*.

Program na průjezd prozkoumanou křižovatkou *run_over_cross* byl jednoduchý, v podstatě jde jen o vjetí na křižovatku (stejné jako u programu *explore_cross*) a zaslání zprávy *„over_cross“*, že byl úkon ukončen. Diagram komunikace mezi MRDS a NXT–G programy viz Obrázek 32 na straně iv v příloze.

Dále bylo potřeba vytvořit několik programů nepřímo souvisejících s aplikací prohledávání bludiště. Byly to programy pro kalibraci, program pro zviditelnění ukončení prohledávání a ukončení nalezení nejkratší cesty. Na kalibraci světelných senzorů jsme využili blok z ukázkového programu na sledování čáry popsaného v [6]. Pro zvýraznění ukončení

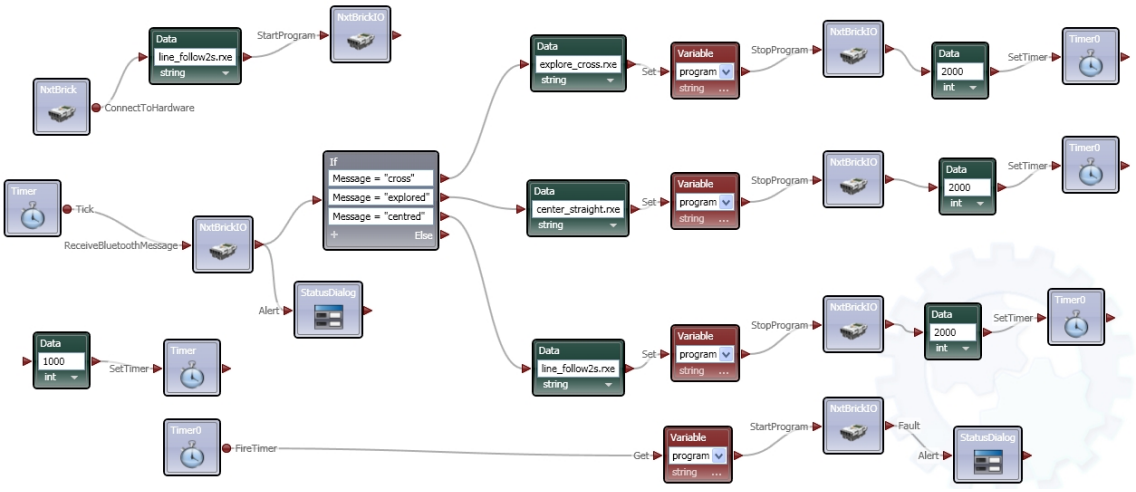
prohledávání a nalezení nejkratší cesty bylo použito několik rotací robota kolem své osy a nakonec přehrání zvuku.

6.2.2 Program ve VPL

Grafický nástroj VPL popsaný v kapitole 4.3 na straně 18 nám posloužil k rychlému vytvoření kostry programu služby pro prohledávání bludiště. Je sice možno si nechat vygenerovat kostru služby v jiném nástroji, ale je vytvořena pouze prázdná struktura služby a všechny ostatní funkce se musí dopsat v C#. Takto jsme už měli velkou část kódu vygenerovanou.

Základ jsme využili prvně ze služby ukázkového příkladu *MsgReadWrite*, z něhož jsme použili službu pro spouštění programů NXT–G v robotovi. Později se ale ukázalo, že po několika cyklech spuštění programu došlo k zablokování kostky. Kostka se tvářila, jakoby v sobě neměla žádný program, přitom po restartu bylo vše v naprostém pořádku. Asi došlo k souběhu volání příkazu na spuštění programů, při kterém došlo k *deadlocku* (zablokování). Jinak si to nedokážeme vysvětlit. Službu pro spouštění programů jsme nahradili službami MRDS pro NXT a k problému již dále nedocházelo.

Základní program měl umět pouze rozhodnout o spuštění dalšího programu na základě přijaté zprávy. Veškerou komplikovanou logiku prohledávání nám přišlo jednodušší napsat v C#, než ve VPL. Proto jsme vytvořili základ programu, vyzkoušeli zda se chová správně a nechali jej zkompilovat do C#. Program ve VPL viz Obrázek 28. Výsledný program byl poté ale mírně upraven v C# (přibyla větev pro zprávu *„over_cross“* a logika celé aplikace), proto toto schéma úplně neodpovídá realitě.



Obrázek 28: MRDS VPL – řešení bludiště

6.2.3 Program v C#

Úprava pomocí C# hrála klíčovou roli, musela se v něm vytvořit celá logika aplikace – prohledávání bludiště, paměť aktuální pozice a vyhledání nejkratší cesty. Postupně vznikaly třídy, které byly základními stavebními kameny této aplikace.

Nejprve bylo potřeba ukládat informace o prohledaných křižovatkách. Pro pozici robota by sice stačilo dvourozměrné pole, křižovatky by také stačilo ukládat do dvourozměrného pole, kde vnitřní hodnoty by znamenaly tvar křižovatky. Tímto způsobem by se nejspíš řešila implementace celé aplikace do paměti robota v RobotC. K dispozici jsme ale měli plně objektový jazyk C#, proto jsme se rozhodli pro objektovou reprezentaci křižovatek. Vznikla třída *Cross.cs*, pomocí třídních proměnných udržovala reference na okolní křižovatky. Okolní křižovatky byly opět typu *Cross*. Každá křižovatka obsahuje navíc proměnnou typu *Point* s uloženou pozicí, pro pozdější lokaci. Navíc obsahuje *Boolean* informaci o tom, zda-li je křižovatka prozkoumaná. Tak je možno z křižovatky zjistit, zda jsou okolní křižovatky prozkoumány. Pro snadnější hledání chyb jsme křižovatce přidali atribut *Id*, který znamená pořadové číslo objevení křižovatky. Třídní diagram křižovatky viz Obrázek 33 na straně v v příloze.

Dále jsme vytvořili hlavní třídu celé aplikace, která se stará o logiku celého prohledávání. Nazvali jsme ji *Labyrinth.cs*. Vytvořili jsme v ní metody, které budou volány z vygenerované služby. Tato třída obsahuje základní informace jako jsou velikost bludiště, názvy programů NXT–G, které se mají spouštět. Vektory pro směry pohybu, zásobník s cestou procházených křižovatek. Paměť všech objevených křižovatek je zajištěna hash mapou, kde klíčem je pozice křižovatky. Třída taky má v paměti uloženu aktuálně prozkoumávanou křižovatku a aktuální směr pohybu pro správné určení dalšího pohybu a další důležité informace, jako je pozice startovní a cílové křižovatky, viz Obrázek 33 na straně v v příloze.

Z vygenerovaného kódu jsme si našli část, kde bylo potřeba provázat službu s nalezením křižovatky, viz Výpis 13. Stačilo provázat třídu *OnTimerTickHandler*, ve které byla metoda starající se o příjem bluetooth zpráv. Proto jsme v této třídě vytvořili statický objekt třídy *Labyrinth* a nazvali jej *lab*. V původním vygenerovaném kódu po přijetí zprávy *„cross“* se nastavila proměnná programu, který se měl spustit na hodnotu *explore_cross.rxe*. My jsme potřebovali, aby v závislosti na tom, zda je křižovatka prozkoumaná a zda se jedná o cílovou křižovatku, spustil program jiný. Proto jsme v třídě *Labyrinth* vytvořili metodu *cross*, která jako návratovou hodnotu vrací jméno spouštěného programu.

```
void OnReceiveBluetoothMessageSuccess(io.BluetoothMessage response){
    if (response.Message == @"cross"){
      string program=lab.cross();      //volání metody cross třídy Labyrinth
      base.StateChanged = true;
      DiagramState a = new DiagramState();
      State.program = program;
      a.program = State.program;       //nastavení spouštěného programu
      Increment();
      Activate(ccr.Arbiter.Choice(
         NxtBrickIOPort.StopProgram(new io.StopLegoProgramRequest()),
         OnStopLegoProgramRequestSuccess,
         delegate(soap.Fault fault){ base.FaultHandler(fault, @"NxtBrickIOPort.StopProgram(new
              ␣io.StopLegoProgramRequest())");
           Decrement();
         }));}
```

Výpis 13: Část metody ošetřující příjem bluetooth zpráv

Metoda *cross* zjistí, zda-li se jedná o první křižovatku, pokud ano, vytvoří nový objekt typu *Cross* a nastaví jeho souřadnice a *Id*. Pak zapíše do souboru průjezd touto křižovatkou. Pokud byla prozkoumána, vrátí jméno programu na průjezd, pokud nebyla, vrátí jméno programu na průzkum křižovatky. Přenastaví hodnotu aktuální křižovatky na prozkoumaná a uloží informace o křižovatce do souboru (jen pro potřeby ladění). Nakonec proběhne testování, jestli se nejedná o cílovou křižovatku. Pokud ano a není nastaveno nalezení nejkratší cesty, program končí, jinak pokračuje v průzkumu, viz Výpis 14.

```
public string cross(){
 if (lastCross == null){
  path.Push(startPoint);    // vložení křižovatky do zásobníku procházené cesty
  lastCross = new Cross(startPoint, lastId);
  vertexDictionary.Add(startPoint, lastCross); // vložení křižovatky do hash mapy objevených
      křižovatek
 }
 bool exploredBackup = lastCross.Explored;
 lastCross.Explored = true;

 // debug výpis do souboru
 using (StreamWriter f = new StreamWriter(Path.Combine(OUTPUT_PATH, "path.txt"), true))
 {
  f.WriteLine("cross id: " + lastCross.Id + " position: " + lastCross.Position);
 };
 if (exploredBackup){
   return RUN_OVER_CROSS_PROGRAM; //pokud je prozkoumaná pouze průjezd křižovatkou.
 }
 if (lastCross.Position.Equals(endPoint)){
   if (findShortestPath && !goViaShortestPath){
     return EXPLORE_CROSS_PROGRAM;
   }
  return FINISH_PROGRAM; // cílová křižovatka objevena, nehledáme nejkratší cestu
 }
 return EXPLORE_CROSS_PROGRAM;
}
```

Výpis 14: Metoda cross třídy Labyrinth

Při prohledávání bludiště bylo nutné vytvářet objektovou reprezentaci celé topologie. K tomu bylo zapotřebí provázat zachycení příjmu zpráv typu *„straight“*, *„right“* a *„left“*, ty bylo potřeba dále zpracovat. Část metody služby zachytávající zprávy byla popsána výše, viz Výpis 13, tentokrát ale nebudeme zachytávat zprávu *„cross“*, ale všechny tři zprávy o směru a budeme volat metodu *direction* třídy *Labyrinth*, která dostane jako parametr zachycený směr. Metoda nic nevrací, jen zpracuje přijaté informace o směrech křižovatky. Část kódu této metody viz Výpis 15.

Tato metoda v závislosti na směru, jaký byl při příjezdu na křižovatku, dosadí reference křižovatky na směr, který byl zaslán. Původně jsme měli záměr, že proměnné *Straight*, *Right*, *Left* a *Back* třídy *Cross* se budou dosazovat v závislosti na směru, ze kterého robot přijel. Čili např. *Straight* bude vždy znamenat směr rovně tak, jak robot přijel. Nakonec jsme došli k závěru, že bude vhodnější, pokud bude každá křižovatka orientovaná stejně.

Čili např. *Straight* bude vždy nahoru vzhledem k bludišti bez ohledu na směr, kterým robot přijel. Proto musí být ošetřeny všechny směry, zde je vidět jen směr *UP*.

```
public void direction (String direction ){
   //debug vypis
   using (StreamWriter f = new StreamWriter(Path.Combine(OUTPUT_PATH, "path.txt"), true))
   {
   f.WriteLine( direction );
   };
   //pokud je poslední směr jízdy dopředu
   if ( lastDirection == UP){
      // přijato bylo rovně
   if ( direction .Equals("straight") && lastCross.Straight == null){
      lastCross.Straight = processDirection(vectorUp);
      lastCross.Straight.Back = lastCross;
    }
    else if ( direction .Equals("left") && lastCross.Left == null){
      lastCross.Left = processDirection( vectorLeft );
      lastCross.Left.Right = lastCross;
    }
    else if ( direction .Equals("right") && lastCross.Right == null){
      lastCross.Right = processDirection(vectorRight);
      lastCross.Right.Left = lastCross;
    }
   }
   ...
}
```

Výpis 15: Metoda direction třídy Labyrinth

Metoda *processDirection* dostane jako parametr vektor směru, ve kterém se nachází objevená křižovatka a vypočte její souřadnice a vrátí nový objekt této křižovatky.

```
public Cross processDirection (Size direction ){
   // Výpočet souřadnice sousední křižovatky v~tomto směru
   nextPoint = Point.Add(lastCross.Position, direction );
   // Test zda už nebyla objevena
   if (! vertexDictionary .ContainsKey(nextPoint)){
      // Vytvoření nového objektu
      lastId ++;
      Cross nextCross = new Cross(nextPoint, lastId );
      vertexDictionary .Add(nextPoint, nextCross);
      return nextCross;
   }else{
      // Návrat křižovatky z~paměti objevených křižovatek
      Cross oldCross = vertexDictionary[nextPoint];
      return oldCross;
   }
}
```

Výpis 16: Metoda processDirection třídy Labyrinth

Dále bylo potřeba zajistit asi nejdůležitější část, logiku prohledávání. Po prozkoumání křižovatky a příjmu zprávy *„explored“* nebo průjezdu křižovatkou a příjmu zprávy

„over_cross“ se musí robot rozhodnout o směru dalšího průzkumu. Opět bylo nutno provázat metodu příjmu bluetooth zpráv s třídou *Labyrinth*. Pro oba typy zpráv je funkce podobná, vždy se musí vypočíst směr dalšího prohledávání a vrátit jméno programu pro natočení do tohoto směru. Proto obě zprávy navazují na stejnou metodu, pouze mají jiný parametr typu *boolean* prozkoumáno *ANO/NE*. Provázání je s metodou *getCenterProgram*, která stejně jako metoda *cross*, vrací jméno spouštěného programu. Tentokrát jde ale o jeden z programů, který se stará o natočení robota do jednoho ze čtyř směrů. Tato metoda je příliš rozsáhlá na to, abych zde dával její výpis. Proto ji alespoň popíši.

Nejprve proběhne testování zda robot již nejede nejkratší vypočítanou cestou do cíle. V takovém případě se vrátí křižovatka vybraná ze zásobníku vypočítané nejkratší cesty (vytvoření zásobníku s vypočítanou nejkratší cestou bude popsáno později) a z ní se vypočítá směr další cesty. Poté dojde k prověření, jestli sousední křižovatka není cíl. Pokud se nehledá nejkratší cesta, robot pojede přímo k cíli. Pokud nenastane ani jedna z možností, je vybrán jeden z neprozkoumaných směrů v pořadí vzhledem k poloze robota: nejprve rovně, doleva a nakonec doprava. Směr zpět je jistě prozkoumaný, není třeba jej testovat. Pokud jsou všechny tři směry prozkoumány, dochází k vracení robota po cestě, ze které přijel. Proto se ze zásobníku aktuálně procházené cesty vybere naposledy procházená křižovatka před aktuální křižovatkou. Pozn. pokud byla posledně procházená křižovatka slepá ulička, automaticky byla vyjmuta ze zásobníku při návratu zpět a vybere se křižovatka procházená před vyjmutou křižovatkou. Takto je procházením do hloubky prozkoumáváno bludiště. Pokud je nastaveno hledání nejkratší cesty je prozkoumáno celé bludiště a dokud se robot nevrátí na startovní pozici nezačne výpočet nejkratší cesty. Pokud se nehledá nejkratší cesta, pravděpodobně k nalezení cíle nebude potřebovat prozkoumat celé bludiště.

Hledání nejkratší cesty je závislé na prohledání úplně celého bludiště. V opačném případě by mohlo dojít k tomu, že algoritmus nevrátí nejkratší cestu, protože ji robot prostě neobjevil. Takže propočítávání nejkratší cesty začíná v momentu, kdy je robot ve startovní pozici a všechny cesty z ní vedoucí jsou prozkoumané. Pro hledání nejkratší cesty jsme vytvořili třídu *ShortestPath*. Ta obsahuje objekt *Point*, pozici křižovatky a referenci na předešlý *ShortestPath*. Slouží jen jako datová struktura vytvářející zřetězený seznam bodů křižovatek. Tak jsme schopni po procházení do šířky a nalezení cíle zpětně vykonstruovat nalezenou nejkratší cestu. U prohledávání do šířky cest s jednotnou vzdáleností je zaručeno, že nejprve nalezneme cíl tou nejkratší cestou.

Prohledávaní do šířky funguje následovně. Začínáme vložením startovního bodu do fronty. V našem případě jsme vložili do fronty objekt *ShortestPath* obsahující startovní pozici křižovatky, neukazující na žádný předchozí objekt *ShortestPath*. Startovní pozici taky vložíme do množiny zpracovaných bodů. Pak, dokud je fronta neprázdná nebo dokud nenalezneme cíl, vybíráme objekty *ShortestPath* z fronty. Z vybraného objektu pomocí pozice získáme objekt křižovatky. Pokud některá ze sousedních křižovatek není v množině zpracovaných, vytvoříme *ShortestPath* objekt s její pozicí a referencí na vyjmutý *ShortestPath* z fronty. Takto vytvořený objekt přidáme do fronty. Po vložení ji označíme za zpracovanou. Pokračujeme, dokud nenalezneme cílovou křižovatku. V této chvíli máme nejkratší cestu do cíle v obráceném pořadí, tedy z cílové do startovní křižovatky. Proto do zásobníku s nejkratší cestou budeme postupně přidávat objekty z posledního objektu

ShortestPath tak dlouho, dokud odkazuje na další *ShortestPath*. Nakonec máme nejkratší cestu ve správném pořadí. Teď můžeme předat směr k další křižovatce po nejkratší cestě. Tak robot pokračuje až do doby, kdy je zásobník prázdný, a tím pádem se nachází na cílové křižovatce, viz Výpis 17.

```
private void calculateShortestPath (){
    Dictionary<Point, Cross> processed = new Dictionary<Point, Cross>(VRCHOLU);
    Queue<ShortestPath> workingQueue = new Queue<ShortestPath>(VRCHOLU);

    workingQueue.Enqueue(new ShortestPath(startPoint, null));
    processed[ startPoint ] = null;
    ShortestPath processPath = null;
    ShortestPath nextPath = null;
    Cross processCross;
    bool found = false;
    while (workingQueue.Count > 0)
    {
        processPath = workingQueue.Dequeue();
       processCross = vertexDictionary[processPath.LastCrossPoint];
       if (processCross.Straight != null){
            if (!processed.ContainsKey(processCross.Straight.Position)){
            processed[processCross.Straight.Position] = null;
            nextPath = new ShortestPath(processCross.Straight.Position, processPath);
                if (processCross.Straight.Position.Equals(endPoint)){
                    found = true;
                    break;
                }
            workingQueue.Enqueue(nextPath);
            }
        }
        if (processCross.Left != null){
        ...
        }
        if (processCross.Right != null){
        ...
        }
        if (processCross.Back != null){
        ...
        }
        if (found){
            while (nextPath.PreviousCrossPath != null){
                shortestPath.Push(nextPath.LastCrossPoint);
                nextPath = nextPath.PreviousCrossPath;
            }
        }
    }
}
```

Výpis 17: Metoda calculateShortestPath třídy Labyrinth

Pro testování logiky prohledávání jsme nepotřebovali vůbec prostředí MRDS, stačilo vhodně volat metody třídy *Labyrinth* a debuggovat ve Visual Studiu. Postupně jsme přidávali volání metod a sledovali stav proměnných a návratové hodnoty funkcí. Pro velkou

mapu to byla sice časově náročná procedura, ale výsledek byl uspokojivý. Bohužel se tak testovala jen logika aplikace za předpokladu, že NXT–G programy v robotovi budou fungovat správně (např. správně detekuje všechny cesty vedoucí z křižovatky). Simulace prozkoumání několika křižovatek viz Výpis 18.

```
public static void Main(){
    Labyrinth l = new Labyrinth();   //Vytvoření objektu třídy Labyrinth

    string program = l.cross();   //Simulace příjezdu na křižovatku
    Console.WriteLine("ID:␣" + l.lastCross.Id + "␣" + l.lastCross.Position);
    Console.WriteLine(program);
    l.direction("straight");   //Simulace detekce cesty vedoucí rovně
    Console.WriteLine(l.getCenterProgram(false));   //Test správnosti směru dalšího prohledávání

    // Další křižovatka
    program = l.cross();
    Console.WriteLine("\nID:␣" + l.lastCross.Id + "␣" + l.lastCross.Position);
    Console.WriteLine(program);
    l.direction("straight");
    l.direction("right");
    l.direction("left");
    Console.WriteLine(l.getCenterProgram(false));
    ...
}
```

Výpis 18: Debbugování průzkumu pár křižovatek

Takto vytvořená aplikace funguje a je k vidění na video kanálu naší katedry informatiky **<http://www.youtube.com/user/informatikaVSB#p/u/0/CFvaKd2C8Mk>**. Tento kanál založil Ing. Michal Radecký. Jemu taky vděčím za sestřih tohoto videa. Video je dvakrát zrychleno, celkově trvalo prohledávání a nalezení nejkratší cesty něco okolo 10 minut. Je to způsobeno především nutnou prodlevou mezi ukončením a spuštěním NXT–G programů. Při nižších než 1,5s intervalech mezi spuštěním docházelo k tomu, že se program nespustil. Tento nedostatek se nám bohužel doposud nepodařilo odstranit. Původně jsme testovali robota na menším bludišti se 6 křižovatkami, tam nebylo zpoždění příliš znatelné. Při 24 křižovatkách to už ale byla velká časová ztráta. Při konkrétní navržené topologii bludiště musel robot projet více než 50 křižovatkami, než se dostal nejkratší cestou do cíle, a to i přes optimalizaci prohledávání. Při průzkumu projede všechny křižovatky, ale někdy nemusí projet všechny cesty mezi křižovatkami. Zjistí si, jestli je sousední křižovatka prozkoumaná, pokud ano, už k ní touto cestou nejede. Pokud sousední křižovatka má ještě nějaké další neprozkoumané cesty, je zaručeno, že se při prohledávání do hloubky někdy k této křižovatce vrátí a prozkoumá je.

7 Závěr

Cíl této práce byl splněn. Byla implementována aplikace pro robota LEGO Mindstorms NXT v prostředí MRDS popsaném v kapitole 4. Bohužel vznikly problémy v komunikaci mezi MRDS a LEGO NXT popsané v kapitole 6.1, které znemožnily řešení aplikací vyžadující informace ze senzorů v reálném čase. Možná u robotů jiných platforem k tomuto problému v kombinaci s MRDS nedochází. Tento problém byl vyřešen napsáním části aplikace v nástroji LEGO Mindstorms (NXT–G) popsaném v kapitole 2.3. Přesto považuji MRDS sice jako revoluční myšlenku podpory programování velké škály platforem robotů, pro LEGO Mindstorms NXT však dle mého názoru nejlepší nástroj na programování je RobotC popsaný v kapitole 3.

V této práci se nám podařilo vytvořit aplikaci pro LEGO robota, který je schopen řešit bludiště a najde v něm nejkratší cestu. To vše s použitím nástrojů MRDS a LEGO Mindstorms (NXT–G). Prozkoumán byl i velmi zdařilý nástroj RobotC, který hodnotím asi jako nejlepší z těch, co jsem měl možnost poznat. Bohužel se nám nepodařilo odstranit časovou prodlevu mezi spouštěním NXT–G programů. To je námět pro další práci.

Později jsme přemýšleli nad tím, jak omezit časovou prodlevu mezi spouštěnými programy NXT–G. Napadlo nás, že by se vytvořil jeden komplexní NXT–G program, obsahující všechny potřebné programy. Jejich spuštění by pak záviselo na zprávě bluetooth zaslané z počítače. Pak by se spustil NXT–G program pouze jednou, a tím by se odstranila časová prodleva. Spuštění podprogramu po zaslání bluetooth zprávy by bylo téměř okamžité. Nevím ovšem, zda–li by se sjednocením programů do jednoho bloku v konečném důsledku nezvětšila paměťová velikost celého programu. Pak by mohlo nastat to, že se NXT–G program nevejde do paměti robota NXT.

David Turoň

8 Reference

[1] *Active Robots*. [online], 15. 3. 2010, naposledy navštíveno 15. 3. 2010.
URL **<http://www.active-robots.com/>**

[2] *HiTechnic – produkty*. [online], 15. 3. 2010, naposledy navštíveno 15. 3. 2010.
URL **<http://www.hitechnic.com/products>**

[3] *LEGO – LEGO produkty*. [online], 15. 3. 2010, naposledy navštíveno 15. 3. 2010.
URL **<http://www.lego.com/en-US/products/default.aspx>**

[4] *LEGO Mindstorms*. [online], 15. 3. 2010, naposledy navštíveno 15. 3. 2010.
URL **<http://mindstorms.lego.com>**

[5] *Microsoft Robotics*. [online], 15. 3. 2010, naposledy navštíveno 15. 3. 2010.
URL **<http://www.microsoft.com/Robotics>**

[6] *NXT Programs – Line Follower*. [online], 15. 3. 2010, naposledy navštíveno 15. 3. 2010.
URL **<http://www.nxtprograms.com/line_follower/steps.html#Program>**

[7] *RobotC*. [online], 15. 3. 2010, naposledy navštíveno 15. 3. 2010.
URL **<http://www.robotc.net/>**

[8] *Roboti od LEGA a Microsoft Robotics Studio*. [online], 15. 3. 2010, naposledy navštíveno 15. 3. 2010.
URL **<http://www.vyvojar.cz/Articles/429-roboti-od-lega-a-microsoft-robotics-studio.aspx>**

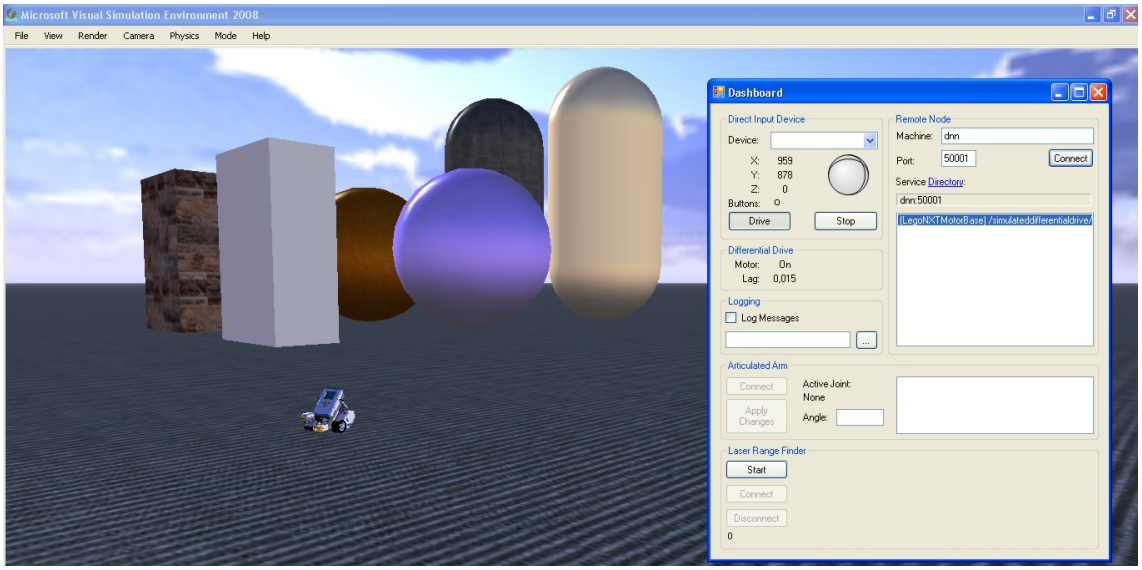
[9] *Wikipedie – Lego Mindstorms NXT*. [online], 15. 3. 2010, naposledy navštíveno 15. 3. 2010.
URL **<http://en.wikipedia.org/wiki/Lego_Mindstorms_NXT>**

[10] Kopecký, L.: *Porovnávání prostředí pro řízení mobilních robotů*. Bakalářská práce, ČVUT, 2009.

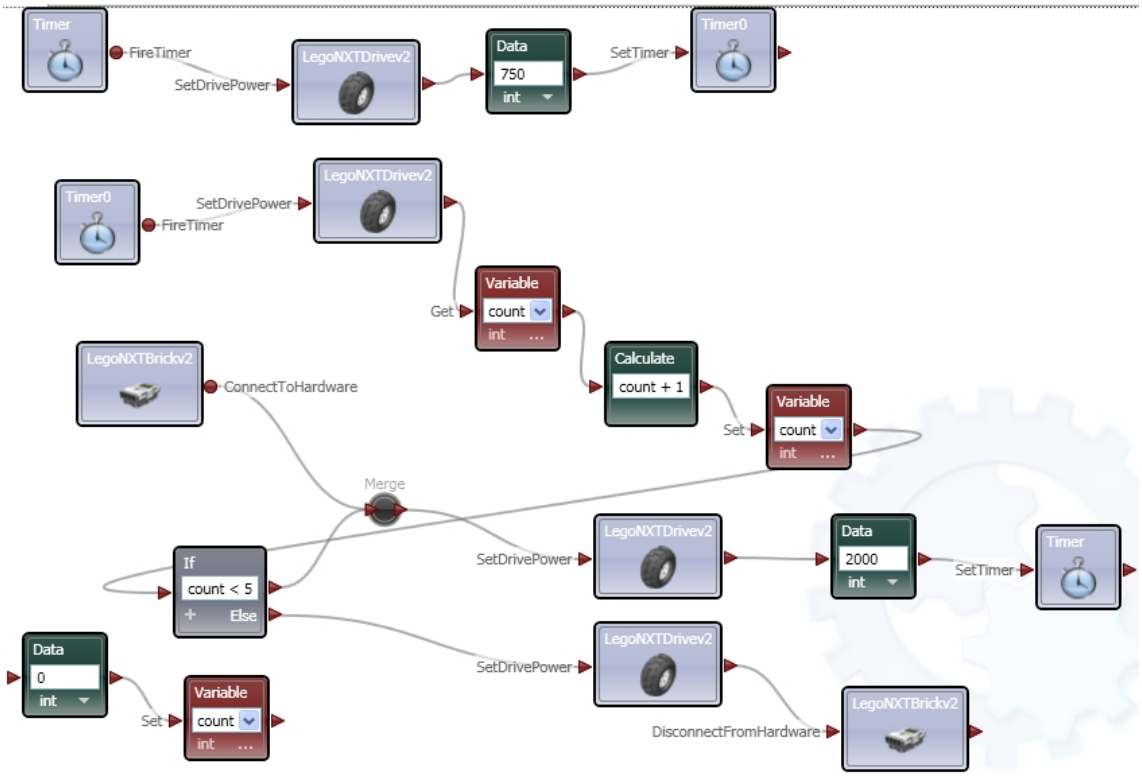
[11] Morgan, S.: *Programming Microsoft Robotics Studio*. Microsoft Press, 2008.

[12] Toledo, S.: *Analysis of the NXT Bluetooth–Communication Protocol*. [online], 15. 3. 2010, naposledy navštíveno 15. 3. 2010.
URL **<http://www.tau.ac.il/∼stoledo/lego/btperformance.html>**

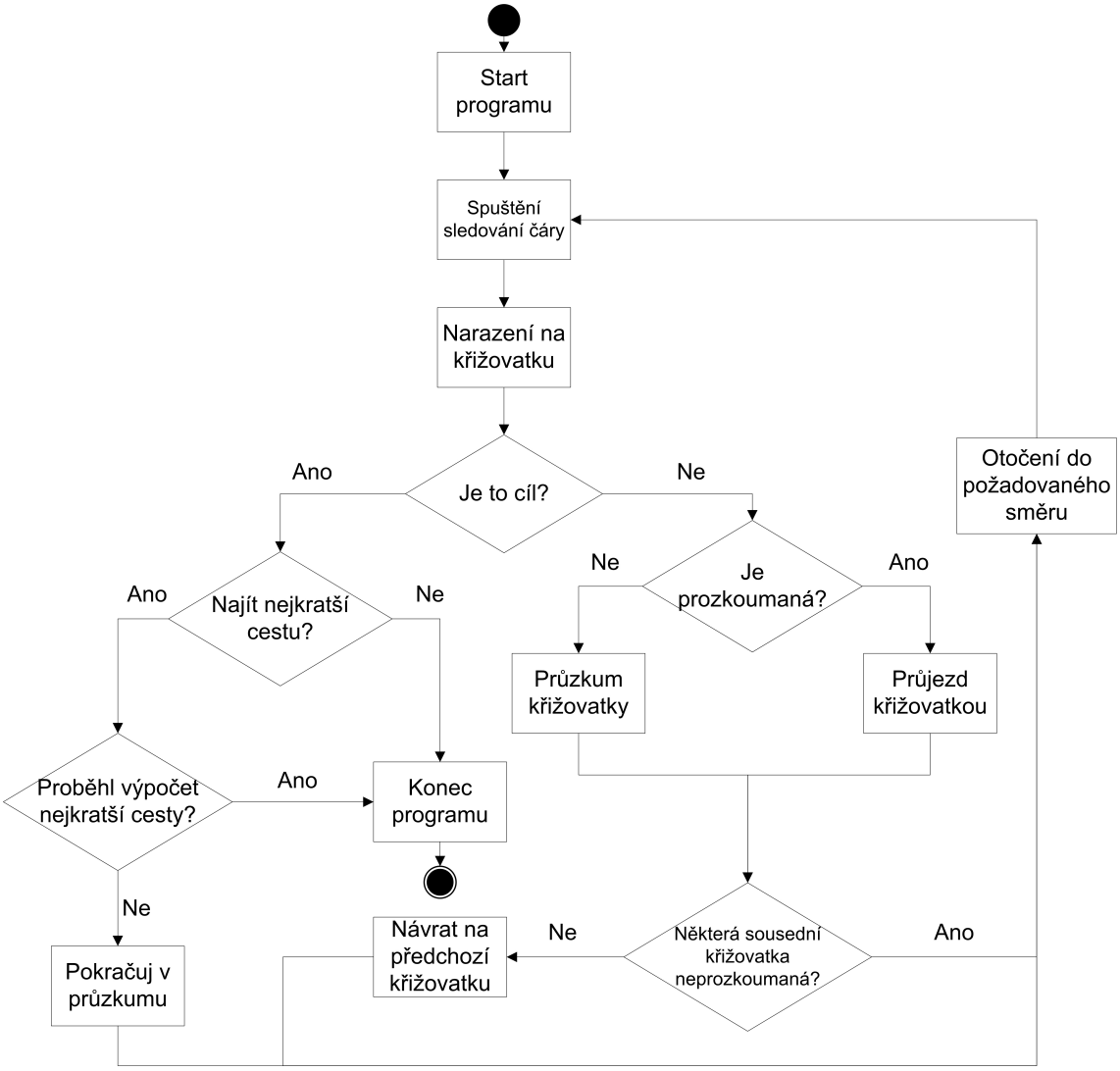
A Obrázky



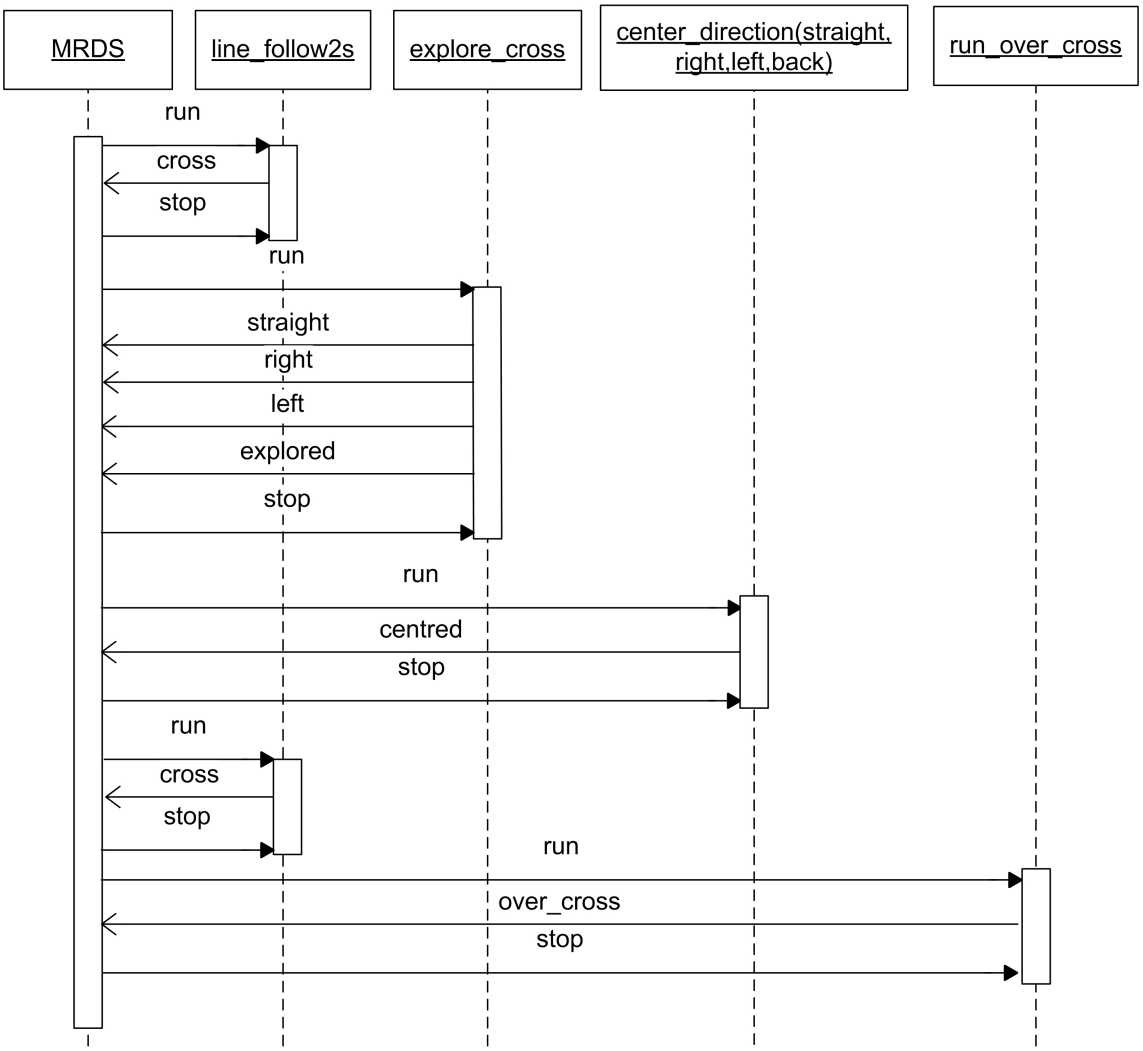
Obrázek 29: MRDS VSE



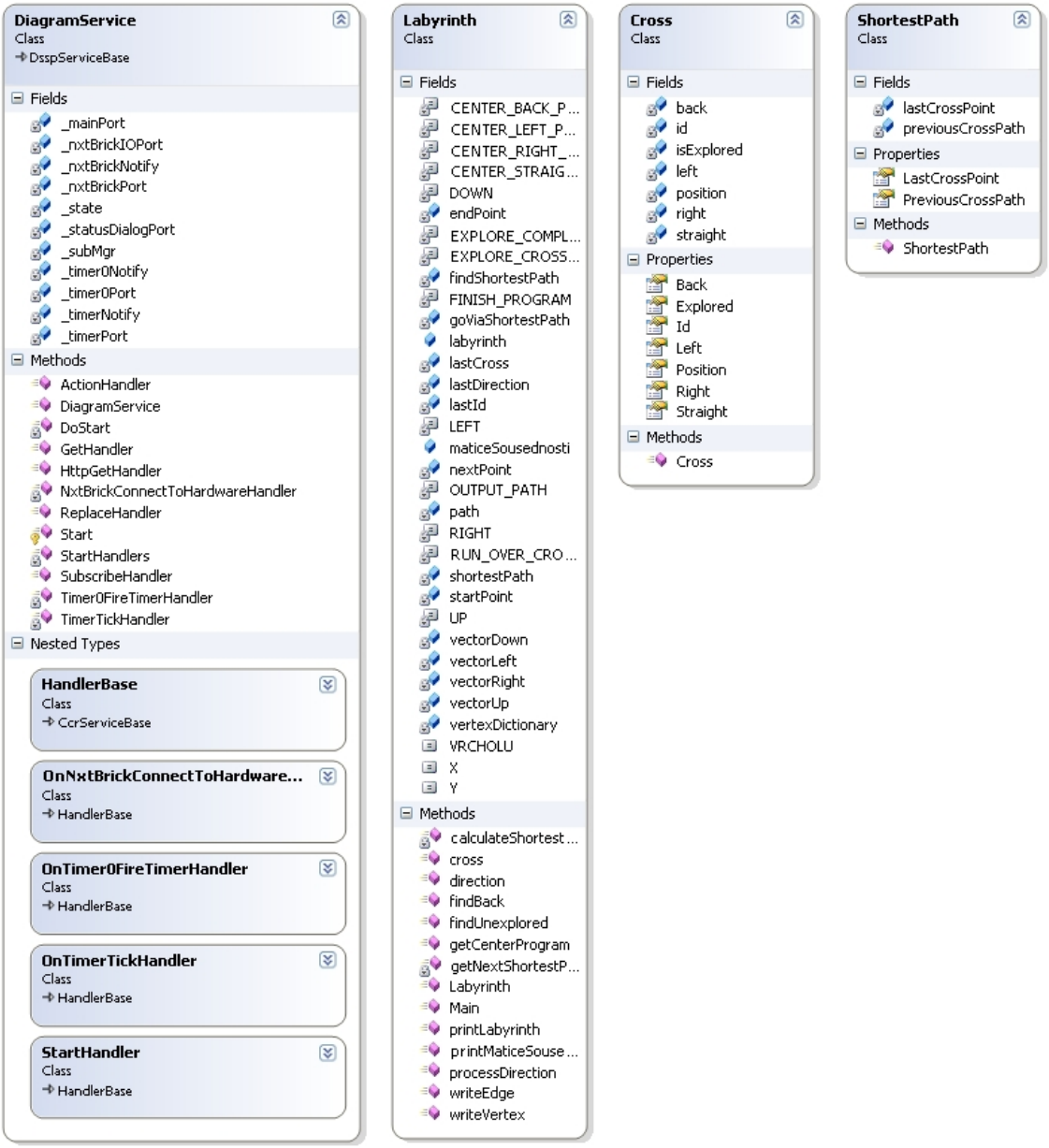
Obrázek 30: Program „nxt_loop_1_while“ ve VPL



Obrázek 31: Diagram aktivit prohledávání bludiště



Obrázek 32: Sekvenční diagram komunikace MRDS s programy NXT–G



Obrázek 33: Diagram tříd – řešení bludiště

B Obsah CD

- Zdrojové soubory aplikace NXT Labyrinth Solver.
- Zdrojové soubory jednoduchých úloh ve VPL.
- Text práce v PDF a LaTeX.